

NASA-CR-193643

JOHNSON
1N-61-CR
186044
51P

Advanced Software Development Workstation

Knowledge Base Methodology - Methodology for First Engineering Script Language (ESL) Knowledge Base

Kumar Peeris
University of Houston-Clear Lake
Michel Izygon
Barrios Technology, Inc.

May 31, 1993

N94-21065

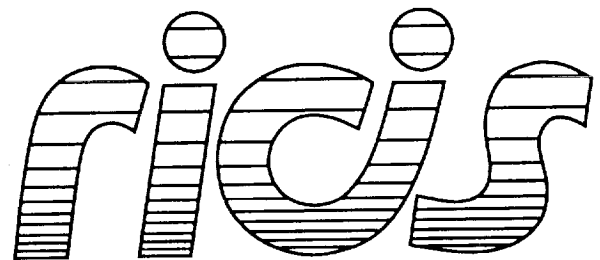
Unclass

G3/62 0186044

Cooperative Agreement NCC 9-16
Research Activity No. SR.02

NASA Johnson Space Center
Information Systems Directorate
Technology Development Division

(NASA-CR-193643) ADVANCED SOFTWARE
DEVELOPMENT WORKSTATION: KNOWLEDGE
BASE METHODOLOGY: METHODOLOGY FOR
FIRST ENGINEERING SCRIPT LANGUAGE
(ESL) KNOWLEDGE BASE (Research
Inst. for Computing and Information
Systems) 51 p



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

FINAL REPORT

Concept

Clear Lake established the Research Institute for Systems (RICIS) in 1986 to encourage the NASA and local industry to actively support research in information sciences. As part of this endeavor, UHCL and JSC to jointly define and manage an integrated advanced data processing technology needed for JSC's administrative, engineering and science responsibilities. In 1986, to jointly plan and execute such research under Cooperative Agreement NCC 9-16, facilities are shared by the two institutions to

to conduct, coordinate, and disseminate research in computing and information systems to government, industry, community and academia. UHCL and its gateway affiliates to research and publications on topics of mutual interest to researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students in Business and Public Administration, Education, Humanities, and Natural and Applied Sciences. Industry in a companion program. This program research and advanced development needs of

relationships with other universities and research common research interests, to provide additional conduct needed research. For example, UHCL partnership with Texas A&M University to help education programs, while other research the "gateway" concept.

to find the best match of sponsors, researchers advance knowledge in the computing and information jointly with its sponsors, advises on research needs for conducting the research, provides technical support to coordinate the research and integrates results of UHCL, NASA/JSC and industry.

The RICIS Concept

The University of Houston Computing and Information Science Center (UHCL) and Johnson Space Center (JSC) in the computing and information sciences. UHCL proposed a partnership with JSC to conduct a program of research in advanced data processing technology. The main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a partnership with UHCL beginning in March 1986 through RICIS. Additional computing and educational facilities are shared to conduct the research.

The UHCL/RICIS mission is to provide advanced and professional level education and research to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and JSC to develop materials, prototypes and conduct research to its sponsors and researchers. The program is implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry and government. The program is focused on serving the needs of the government, industry, community and academia.

Moreover, UHCL establishes a partnership with research organizations, having advanced technical sources of expertise to help conduct research. UHCL has entered into a special agreement with JSC to oversee RICIS research and development. Organizations are involved in the program.

A major role of RICIS then is to provide advanced and research objectives to advance the information sciences. RICIS, working with its sponsors, needs, recommends principal technical and administrative support. The program integrates technical results into the government, industry, community and academia.

Advanced Software Development Workstation

**FINAL REPORT:
Knowledge Base Methodology -
Methodology for First Engineering Script Language (ESL)
Knowledge Base**

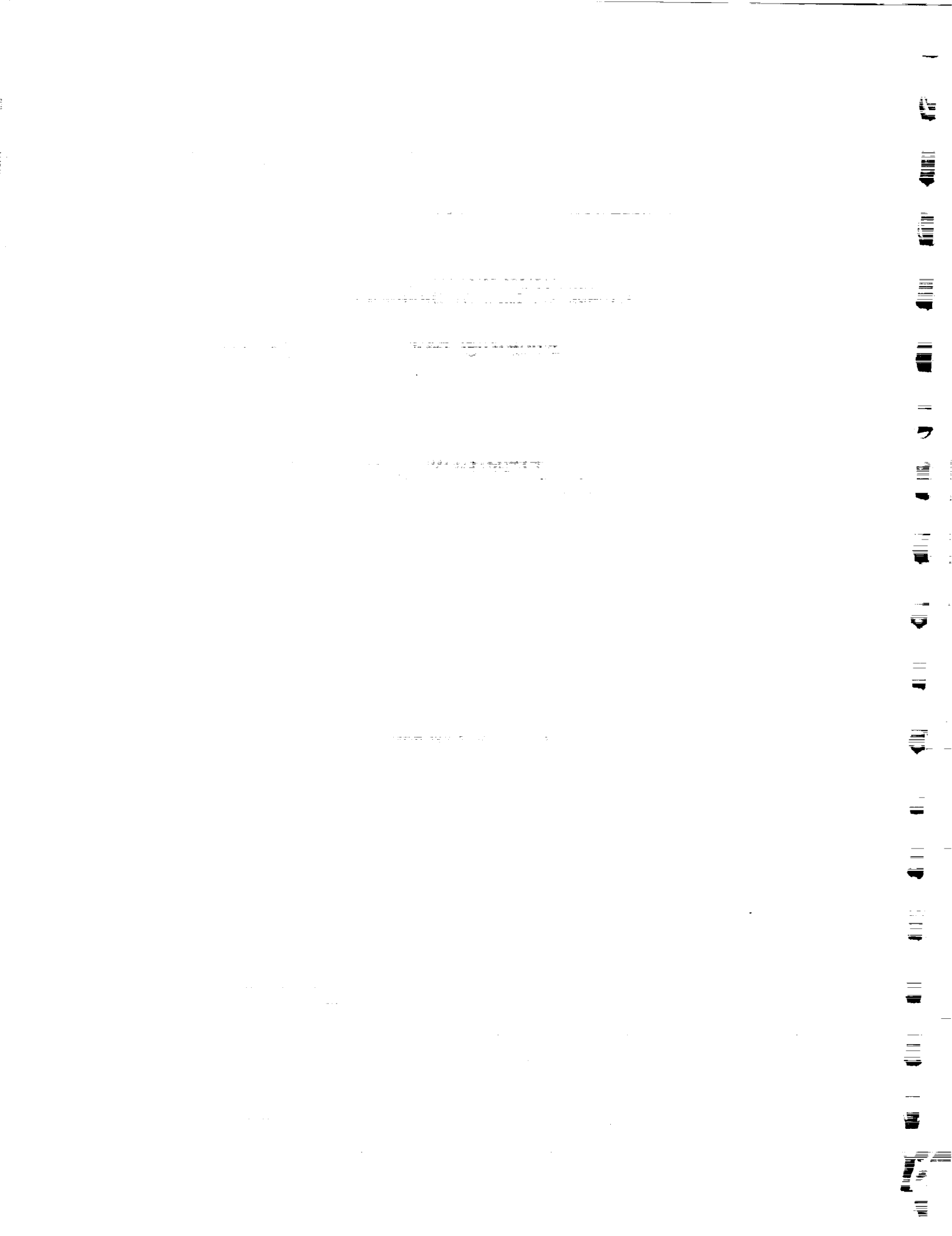
**Prepared for
NASA-Johnson Space Center**

May 31, 1993

**Submitted by
Dr. Michel Izygon
Barrios Technology Inc.
1331 Gemini Av.
Houston, TEXAS 77058**

ABSTRACT

This report explains some of the concepts of the ESL prototype, and summarizes some of the lessons learned in using the prototype for implementing the Flight Mechanics Tool Kit (FMToolKit) series of Ada programs.



RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Kumar Peeris of the University of Houston-Clear Lake and Dr. Michel Izygon of Barrios Technology, Inc. Dr. Rodney L. Bown served as the RICIS research coordinator.

Funding was provided by the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Ernest M. Fridge III, Deputy Chief of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of UHCL, RICIS, NASA or the United States Government.

1. The first part of the document is a letter from the President of the United States to the Congress, dated January 3, 1862. It is a very important document, as it contains the President's annual message to Congress. The letter is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

2. The second part of the document is a report from the Secretary of the Interior, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

3. The third part of the document is a report from the Secretary of the Treasury, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

4. The fourth part of the document is a report from the Secretary of the War, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

5. The fifth part of the document is a report from the Secretary of the Navy, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

6. The sixth part of the document is a report from the Secretary of the State, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

7. The seventh part of the document is a report from the Secretary of the War, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

8. The eighth part of the document is a report from the Secretary of the Navy, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

9. The ninth part of the document is a report from the Secretary of the State, dated January 3, 1862. It is a very important document, as it contains the Secretary's annual report to the President. The report is written in a formal, dignified style, and it is one of the most important documents in the history of the United States.

CONTENTS

Reuse: Background and Concepts.....	1
1.1 Description of the ESL Reuse Method.....	4
1.2 Methods used to reengineer FM tool kit code to ESL Reusable Method.....	13
1.2.1 Analysis of FM Tool Kit Applications.....	14
1.2.2 Modifications and Decomposition of Primitives.....	24
1.2.3 Packages COMMON_MODULES, DATA_TYPES, and DATA_TYPES_SPEC.....	25
1.2.4 Further Modifications.....	26
1.2.5 Modification of procedure COMPUTE_TRAJECTORY_DATA....	29
1.2.6 Application BEST1WAY.....	38
1.2.7 Application POWRSWNG.....	38
1.2.8 Module DATA_MATRIX_INTEGER.....	38
1.2.9 Complete ESL object graphs for all four applications.....	38
2.1 Lessons Learned About Current ESL Tool.....	43

FIGURES

1.1.1	Building an application program with the help of a Parts Composition System.....	7
1.1.2	A typical ESL graph.....	9
1.1.3	Hierarchical decomposition of ESL graphs.....	9
1.1.4	An ESL graph example.....	11
1.1.5	ESL subgraph for INNER_LOOP.....	11
1.2.1	Main program for INTRPLAN.....	14
1.2.2	Main program for IPCAPTUR.....	15
1.2.3a	Modified main program for IPCAPTUR.....	15
1.2.3b	Modified main program for INTRPLAN.....	16
1.2.3c	Procedure INNER_LOOP.....	16
1.2.4	Modified main program for IPCAPTUR.....	17
1.2.5	Added computational statements inside the WHILE loop.....	17
1.2.6a	Modified main program for IPCAPTUR.....	18
1.2.6b	Modified main program for INTRPLAN.....	18
1.2.7	Procedure MAIN_LOOP.....	18
1.2.8	ESL graphical representation of the main program for INTRPLAN or IPCAPTUR.....	19
1.2.9	Modified procedure MAIN_LOOP.....	19
1.2.10	Procedure SET_CONTROL.....	20
1.2.11	Modified procedure INNER_LOOP.....	20
1.2.12	The ESL object graph for subprogram MAIN_LOOP.....	21
1.2.13	The ESL object graph for subprogram INNER_LOOP.....	21
1.2.14	ESL object graph for main program of POWRSWNG.....	22
1.2.15	The ESL object graph for subprogram MAIN_LOOP.....	23
1.2.16	The ESL object graph for subprogram INNER_LOOP of POWRSWNG.....	23
1.2.17	COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET.....	26
1.2.18	COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET.....	26
1.2.19	COMPUTE_POSITION_AND_VELOCITY_OF_SWINGBY_PLANET.....	27
1.2.20	COMPUTE_POSITION_AND_VELOCITY_OF_PLANET.....	27
1.2.21	ESL Object graph for subprogram MAIN_LOOP.....	28
1.2.22	ESL object graph for subprogram INNER_LOOP.....	29
1.2.23	Procedure COMPUTE_TRAJECTORY_DATA.....	30
1.2.24	Procedure EXCEPTION_HANDLER_10001.....	31
1.2.25	Modified Procedure COMPUTE_TRAJECTORY_DATA.....	32
1.2.26	ESL object diagram for COMPUTE_TRAJECTORY_DATA.....	33
1.2.27	COMPUTE_PLANETOCENTRIC_DEPARTURE_DATA.....	34
1.2.28	COMPUTE_PLANETOCENTRIC_ARRIVAL_DATA.....	34
1.2.29	COMPUTE_PLANETOCENTRIC_ARRIVAL_OR_DEPARTURE_DATA..	35
1.2.30	COMPUTE_TRAJECTORY_FOR_FIRST_HELIOCENTRIC_LEG.....	36

FIGURES (cont'd.)

1.2.31	COMPUTE_TRAJECTORY_FOR_SECOND_HELIOCENTRIC_LEG.....	37
1.2.32	COMPUTE_TRAJECTORY_FOR_FIRST_AND_SECOND_ HELIOCENTRIC_LEG.....	37
1.2.33	ESL object graph for POWRSWING.....	39
1.2.34	ESL object graph for INTRPLAN.....	40
1.2.35	ESL object graph for IPCAPTUR.....	41
1.2.36	ESL object graph for BEST1WAY.....	42

**FINAL REPORT:
KNOWLEDGE BASE METHODOLOGY -
METHODOLOGY FOR FIRST ENGINEERING SCRIPT LANGUAGE
(ESL) KNOWLEDGE BASE**

by
Kumar Peeris (UHCL) and Michel E. Izygon (Barrios)

Reuse : Background and concepts.

Software Reuse is one of the technologies that is currently presented as being able to solve the so-called "Software Crisis". In this section, we will describe some of the key concepts of this technology, the different approaches to reusability, some of the issues related to it, and we will try to present how the Engineering Script Language (ESL) implement the reuse paradigm.

Reuse Concepts

The primary goal of reusing software components is that software can be developed faster, cheaper and with higher quality. Though, reuse is not automatic and can not just happen. It has to be carefully engineered. For example a component needs to be easily understandable in order to be reused, and it has also to be malleable enough to fit into different applications. In fact the software development process is deeply affected when reuse is being applied. During component development, a serious effort has to be directed toward making these components as reusable as possible. This implies defining reuse coding style guidelines and applying them to any new component to create as well as to any old component to modify. These guidelines should point out the favorable reuse features and may apply to naming conventions, module size and cohesion, internal documentation, etc... . During application development, effort is shifted from writing new code toward finding and eventually modifying existing pieces of code, then assembling them together. We see here that reuse is not free, and therefore has to be carefully managed.

Approaches to Reuse

There are two different approaches to reusing software components: Adaptive Reuse and Compositional Reuse. Their characteristics are as follow:

- Adaptive Reuse

With this approach, components are templates or patterns and are changed each time they are used.

- **Compositional Reuse**

components are atomic and don't change when they are used.

Issues / Dilemmas

The operational problems of reusability are the following:

- finding components
- understanding components
- modifying components
- composing components

When a programmer has to develop a piece of code, the first thing he does is to look in the library of available components to check if there is one that matches his needs. This search process needs to be able to find not only the exact match, but also the "close enough" type of components if the ideal one does not exist. The difficulty of this step is directly linked to the breadth of the library of components. The more specific are the components, the more numerous they will be in the library, and the more difficult it is to find the appropriate one. This aspect of the reuse process is dealt with library systems.

Understanding a component is the next step the programmer will go through, in order to be able to use properly the component he found during the search process. If modifications are necessary, i.e. if the component does not match exactly the need of the programmer, the understanding is even more important as he will need to enter into the code and change it. For this understanding process to be successful, there needs to be a lot of emphasis on documentation during design and coding of any reusable component.

Modification of components is the step that seems to be the less automatizable. The programmer has to do his work at customizing the component to his needs. The issue that is related to this step is that we can foresee that many components may be spawned out of a common root component in order to customize it to the different needs of different programmers. The only way to prevent the library to get out of control is to build components that are generic enough to be applied to many different situation.

Composing components is the step that is completely specific to the reuse-based software development process. Once all the needed components have been found, eventually modified, or developed from scratch, there needs to be a framework where the programmer

can specify how to compose these components together to build the targeted application.

ESL vs Reuse

The ESL concept of Reuse is based on the following principles: ESL is targeted toward domain specialists who do not have a sufficient knowledge of Ada programming language to develop code in their domain. The tool would allow them to graphically develop an application from the available pieces of code stored in a software component library. We should point out here, that the ESL system does not address the issue of developing the elementary components that are populating the library. It takes as a first assumption that these components exist, that they are medium to gross grain components written in Ada, and that they were input in the knowledge base with the proper amount of information to allow their retrieval and their correct use. Based on these assumptions, ESL contains the different mechanisms that allows an application developer to build the program he needs from the stored components. Let us now focus on the different parts of ESL:

- The first part of ESL deals with the storage of the components. The system is built on a knowledge base written in ART-IM. This knowledge base contains the important information about the components such as what it does, what the inputs and outputs are, if the component is composed of other components or if it is an elementary one.
- The second part of ESL addresses the issue of retrieving a component. ESL has a Case Base Reasoning (CBR) engine that allows to query the library for components having some similarities with the needed component. The system will present a list of components belonging to the same class, ordered according to the number of identical attributes values. The application developer can refine his query by analyzing the closest component, changing the unfit attributes and then resubmitting the query with the added information.
- The third part of ESL focus on assembling the retrieved components in order to build an application. A graphical editor allows the application developer to graphically link the desired components.
- The fourth part of ESL is the code generator. From the graphical representation of the flow of inputs and triggers through the different components, ESL generates an Ada main program that contains the calls to the different routines chosen by the user.

1.1 Description of the ESL Reuse method

The Engineering Script Language (ESL) is a language designed to allow non programming users to write High Order Language (HOL) programs by drawing directed graphs to represent the program and having the system generate the corresponding program in HOL. For the implementation of ESL proposed, the HOL code to be generated will be Ada.

The building blocks for directed graphs are nodes and connectors. Nodes are visually represented as labeled icons (e.g., rectangles or circles) and have input and output ports which are used to receive produce data. On a graph, an output port from one node may be connected to an input node on another node via a connector. Visually, all connectors passing data between two nodes are represented as a single arrow connecting the icons representing the nodes. In addition, a graph itself can have input ports and output ports which are connected to ports or nodes on the graph. Visually, the set of all graph input ports is represented by a single icon on the left of the editor window. Each arrow from this icon to a node on the graph represents a group of connectors. Similarly, the set of all graph output ports is represented visually as a single icon on the right of the editor window.

Each node on a graph may represent a primitive procedure or function in the HOL (i.e., a primitive subprogram), an ESL control or data-passing mechanism, or another graph. When a node is a primitive subprogram node, the node's ports represent the subprogram's parameters and, if applicable, its return value.

Node Objects

There are several classes of node objects: the subprogram node, (which includes procedure-node, function-node, and subgraph-node objects), the Merge node, the Replicator node, and the control nodes (If, Select, and Iterator).

A subprogram-node object is used to represent a procedure or function coded in the HOL or to represent a graph previously created through the ESL editor. Each subprogram-node object points to a

subprogram object. Subprogram objects are objects visible through the ACCESS tools panel and included in the ACCESS taxonomy.

Subprogram objects have corresponding ports. Ports of a procedure or a function object represent parameters of the corresponding procedure or function or the return value of the function. Ports of a graph object, called graph ports, are mapped to ports on nodes of the graph by connector objects.

Implementation Objects

An implementation object contains information about how a subprogram object is implemented. The merge, replicator, If, Select and Iterator nodes each have an implicit implementation and do not have an associated implementation object. There are three classes of implementation objects.: In line, separately compiled procedure, and package.

In line implementation objects are appropriate only for graph objects. This type of implementation means that when a subgraph node is part of a larger graph for which code is generated, the code corresponding to the subgraph node is generated online.

Implementation objects whose type is separately compiled procedure are valid for all subprogram objects. Such an implementation object indicates that the subprogram is implemented as a separately compiled Ada procedure. For a separately compiled procedure to be called by an Ada program, the program must be first "with" the procedure; then the procedure may be called.

Package implementation objects are valid for subprograms of procedure or function type. Such an implementation object indicates that the subprogram has been implemented as a visible function in an Ada package. For a procedure or function in a package to be called by an Ada program, the program must first "with" the package; then the procedure may be called using the "package.procedure" notation.

Object Hierarchy

The following is the hierarchy of objects in ESL system.

- subprogram
 - primitive subprogram
 - function
 - procedure
- graph
- node
 - subprogram node
 - primitive subprogram node
 - procedure node
 - function node
 - subgraph node
 - merge node
 - replicator node
 - control node
 - if node
 - select node
 - iterator node
- port
 - graph port
 - procedure port
 - function port
 - node port
- connector group
- connector
 - implementation
 - in line implementation
 - separately compiled procedure implementation
 - package implementation
- data type

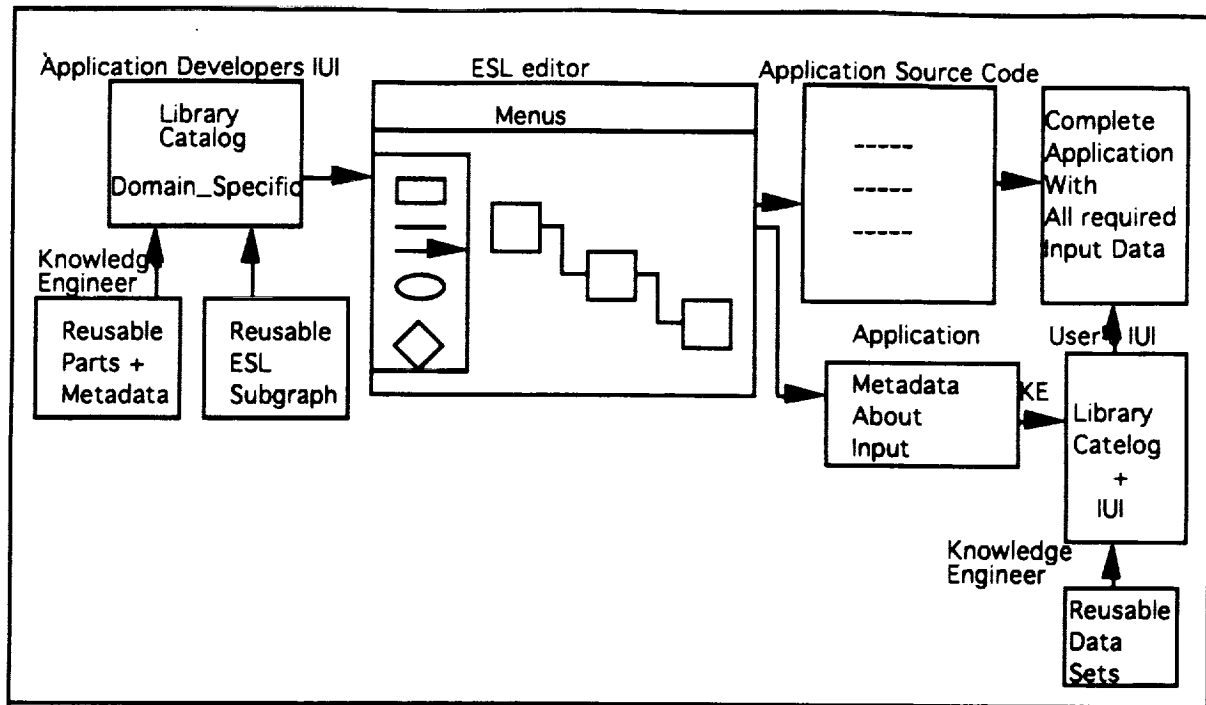


Fig. 1.1.1 Building an Application Program with the help of a Parts Composition System

Figure 1.1.1 shows the various steps that would be involved in building a complete application with the help of a Parts Composition System (PCS), as currently envisioned. A library of procedures (or more generically, primitives) containing software parts that are needed by most application programs within the domain of interest is opened and scanned. If this library contains most of the required primitives, then the application developer may select to use it; otherwise, additional libraries may be searched.

Depending on the decisions of the libraries' management organizations, application developers may or may not be allowed to create modified versions of primitives in the libraries. However, the development, organization, and maintenance of these domain-specific libraries is primarily the responsibility of the software development engineers and not the job of the application developers, who may well be aerospace engineers with minimal programming experience. The software development engineers receive part specifications from the application developers and provide implementations to populate required libraries. If well managed, this separation of roles helps to limit the amount of domain expertise that the software engineer must have and also the amount of programming experience that the application developer must have.

The construction of primitives can be done using the Computer-Aided Software Engineering (CASE) tools. However, a useful, well-maintained library of reusable parts consists of more than a disorganized jumble of parts. A librarian and library tools are clearly required. A librarian must build and maintain a PCS knowledge base using tools that extract the necessary metadata from each primitive (such as input, output, purpose, and constraints) and then catalog this information with the knowledge base's schemas. The cataloging process includes the assignment of each primitive to a specific knowledge base class. Careful development of a meaningful class structure is essential to the usefulness of the library's catalog and one of the most challenging tasks of the knowledge engineer. Special displays may also be required for some classes of primitives in order to make the catalog as user friendly as possible. In short, the knowledge engineer must build an IUI for each domain-specific library of reusable parts. His/her role is to serve as the intermediary between the software development engineers and the application developers.

Once an application developer has selected the most appropriate domain-specific library of parts, he/she invokes the ESL editor. As already explained, the ESL editor allows the application developer to create, modify, store and retrieve graphs that represent applications. The graphs show the structure of an application and what data controls and constraints flow between the components (fig.1.1.2). The components are depicted by boxes called nodes, and the data controls, and constraints are shown as arrows linking the nodes. Other structures, also called nodes, allow for merging and replicating links and for including looping and branching logic. Each component (box) is either a primitive or a subgraph, which makes possible hierarchical decomposition. (fig.1.1.2)

With ESL editor, an application developer uses a mouse and pointer to select menu and palette commands and to select nodes and links on the screen. In this way, graphs are constructed, modified, and stored for possible reuse.

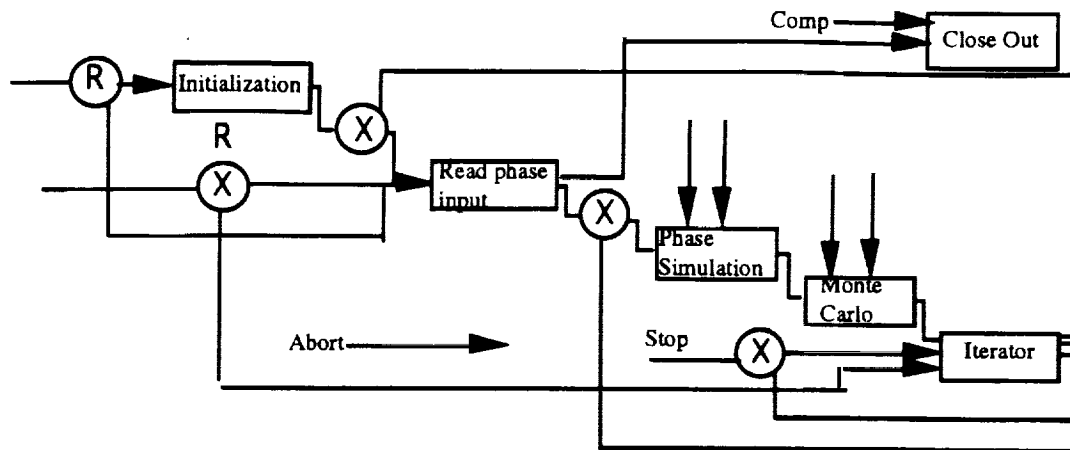


Fig.1.1.2 A typical ESL Graph

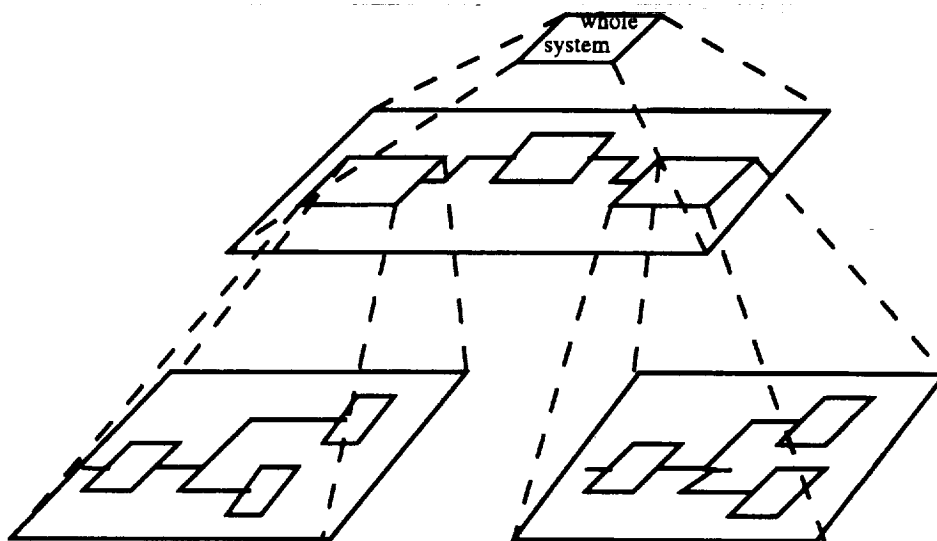


Fig. 1.1.3 Hierarchical Decomposition of ESL graphs

Once the graphs representing an application are completed, the application developer will invoke menu commands to validate the graph system and to generate the required code in some high order language, such as Ada. The generated code, in the form of a main program and subprograms, will then be ready to be compiled and linked with the object code of the primitives from the domain specific library(ies). Alternatively, source code templates (such as Ada generics or even main programs with certain parameters that must be initialized before compilation) might be generated, if required.

ESL graphs will be stored in a knowledge base, where they will be represented, using a schema system, as objects with attributes. The ability to store and retrieve ESL graphs implies a need for well-organized, domain-specific libraries of graphs with good library catalogs. Just as in the case of the libraries of primitives, a knowledge engineer will need to create IUI s for the ESL graph libraries.

The internal representation and storage of graphs, the semantic interpretation and validation of the graphs, and the generation of code in high order language are done using knowledge-based technology.

Graph Implementation and Execution

Fig 1.1.4 depicts a typical example graph created using the ESL editor panel. Each box is an instance of an object. In other words each box is merely a procedure call or a function call. The iterator node indicates an iteration at that particular point until a certain condition is satisfied. INNER_LOOP is a sub graph attached to the main graph. It is a separately edited graph. The sub graph is shown in fig. 1.1.5.

Prior to executing a complete application, the graphs must be translated to a high order language (HOL) representation and subsequently compiled. A graph implementation is an HOL representation of a hierarchical ESL data flow graph that can be compiled by a standard HOL compiler for subsequent execution. The translation process generates the graph implementation by mapping the features found in the application's graph schemas to predefined HOL constructs.

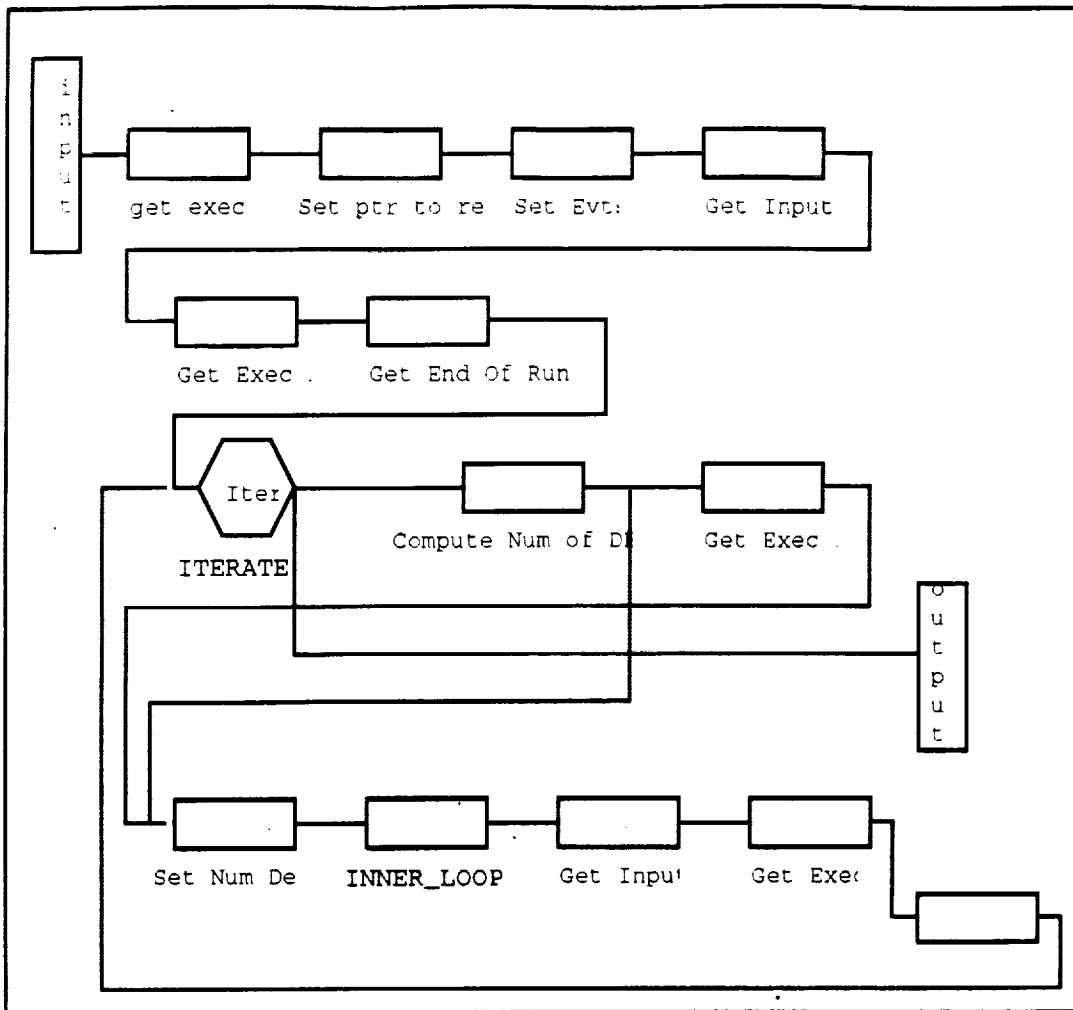


Fig. 1.1.4 An ESL graph example

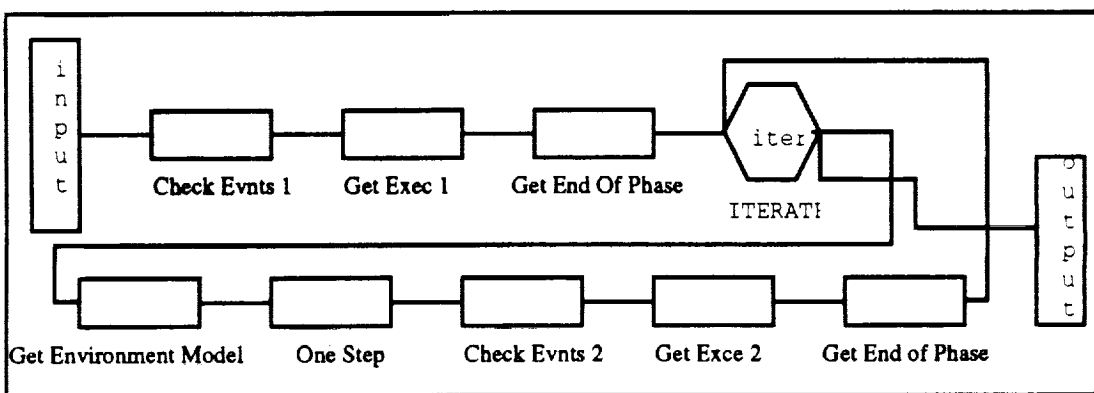


Fig. 1.1.5 ESL sub graph for INNER_LOOP

Generated Code

```
--  
-- Ada code for graph six_dof_driver  
--
```

```
with ASDS_Exec_Record_Manager;  
with six_dof_driver_inner;  
with Environment_model;  
with Six_Dof_Instantiations;  
With state_types;
```

```
procedure six_dof_driver is
```

```
    TEST18 : Boolean := TRUE;
```

```
    Exec15 : ASDS_Exec_Record_Pointer_type;  
    Exec20 : ASDS_Exec_Record_Pointer_type;  
    Num_Diff_Eq16 : Positive;  
    Exec17 : ASDS_Exec_Record_Pointer_Type;  
    Exec19 : ASDS_Exec_Record_Pointer_Type;
```

```
begin
```

```
    -- Code for node Get Exec 1
```

```
    Exec15 := Six_DOF_Instantiations.Get_Exec;
```

```
    -- Code for node Set ptr to rec
```

```
    ASDS_Exec_Record_Manager.set_pointer_to_ASDS_EXEC_record(Exec15);
```

```
    -- Code for node Set Evts
```

```
    Six_DOF_Instantiations.Sst_Discrete_Events;
```

```
    -- Code for node Get Input 1
```

```
    Six_DOF_Instantiations.Six_DOF_INPUT.Get_Input;
```

```
    -- Code for node Get Exec 2
```

```
    Exec19 := Six_DOF_Instantiations.Get_Exec;
```

```
    -- Code for node Get End of Run 1
```

```
    Test18 := Six_DOF_Instantiations.Get_End_Of_Run(Exec19);
```

```
    -- Code for ITERATE
```

```
    while (TEST18) loop
```

```
        -- code for node Compute Num of DEs
```

```
        Num_Diff_Eq16 := State_Type.Compute_Num_Of_Diff_Et;
```

```
        -- Code for node Get Exec 3
```

```
        Exec17 := Six_DOF_Instantiations.Get_Exec;
```

```
        -- Code for node Set Num DEs
```

```
        Six_DOF_Instantiations.Set_Num_Diff_Eq(Exec17, Num_Diff_Eq16);
```

```

-- Code for node inner loop
Six_dof_driver_inner;

-- Code for node Get Input
Six_DOF_Instantiations.six_DOF_Input.Get_Input;

-- Code for node Get Exec
Exec20 := Six_DOF_Instantiations.Get_Exec;

-- Code for node Get End of Run
TEST18 := Six_DOF_Instantiations.Get_End_Of_Run(Exec20);
end loop;
end six_dof_driver;

```

1.2 Methods used to reengineer FM tool kit code to ESL Reusable Method.

As described in section 1.1, we know that the code generated by a designed graph in the ESL system, would be either a main program or a sub program. Also we have mentioned , that a main program or a sub program can be a single procedure or a function call or a set of procedure or function calls or a set of procedure and function calls. In addition, a main program or a sub program can have loop structures and if-then-else structures. An important point is that, ESL does not support nested loop structures. This is one of the limitations provided in the ESL system. Hence, primarily, we need to realize that, reengineering any application should be done within this limited ESL framework.

Currently FM tool kit said to have eleven applications. These source code have been developed in Ada. These applications look very similar. For our "reengineering-for-ESL" purposes, four of these applications - namely INTRPLAN, IPCAPTUR, BEST1WAY and POWRSWNG , have been randomly selected. A vital part of the "reengineering-for-ESL" process is to develop a library of procedures (or more generally, PRIMITIVES) containing the reusable software

components , so that they can be put together to form a complete application.

Analysis of FM-Tool kit applications

Initially, let us consider the two applications INTRPLAN & IPCAPTUR. The code shown below (Fig. 1.2.1 & Fig. 1.2.2) depicts the main programs of the above two applications.

```

with INTRPLEC ; use INTRPLEC ;
with INTRPLIO ; use INTRPLIO ;

procedure INTRPLAN is
begin
RETRIEVE_PREVIOUS_INPUTS_FROM_DISK ;
LET_USER_EDIT_INPUT_DATA ;
SAVE_EDITED_INPUTS_ON_DISK ;
SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
DISPLAY_DATA_SHELL ( NOMINAL_DEPARTURE_DELTA_V ) ;
for J in 0..10 loop
    COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
    for I in 0..16 loop
        COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET ( I ) ;
        COMPUTE_TRAJECTORY_DATA ( I, J ) ;
        DISPLAY_VALUE ( NOMINAL_DEPARTURE_DELTA_V , I, J ) ;
        CHECK_FOR_INTERRUPT_FROM_KEYBOARD ;
    end loop ;
end loop ;
DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;
end ;

```

Fig 1.2.1 - Main Program for INTRPLAN

```

with IPCAPTEC ; use IPCAPTEC ;
with IPCAPTIO ; use IPCAPTIO ;

procedure IPCAPTUR is
begin
RETRIEVE_PREVIOUS_INPUTS_FROM_DISK ;
LET_USER_EDIT_INPUT_DATA ;
SAVE_EDITED_INPUTS_ON_DISK ;
SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
DISPLAY_DATA_SHELL ( NOMINAL_DEPARTURE_DELTA_V ) ;
for J in 0..10 loop
    COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
    for I in 0..16 loop
        COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET ( I ) ;

```

```

        COMPUTE_TRAJECTORY_DATA          ( I, J )          ;
        DISPLAY_VALUE ( NOMINAL_DEPARTURE_DELTA_V , I, J ) ;
        CHECK_FOR_INTERRUPT_FROM_KEYBOARD ;                ;
        end loop ;                                         ;
    end loop ;                                           ;
DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;              ;
end ;

```

Fig 1.2.2 - Main program for IPCAPTUR

The two main programs look exactly the same, except for the different dependent library units. (i.e intrplec & intrplio for INTRPLAN and ipcaptec & ipcaptio for IPCAPTUR). In ESL terms these two are non primitives , because they do not have any computational instructions but a set of module calls. Therefore a major modification is not required except for the elimination of the FOR loops. (In ESL, nested looping structures are not allowed.).

A simple solution to this is to incorporate the inner FOR loop in a separate module and isolate it. Then the two main program structures will look as follows.

```

with IPCAPTEC ; use IPCAPTEC ;
with IPCAPTIO ; use IPCAPTIO ;

procedure IPCAPTUR is
begin
    RETRIEVE_PREVIOUS_INPUTS_FROM_DISK ;
    LET_USER_EDIT_INPUT_DATA ;
    SAVE_EDITED_INPUTS_ON_DISK ;
    SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
    DISPLAY_DATA_SHELL ( TOTAL_DELTA_V ) ;
    for J in 0..10 loop
        COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
        INNER_LOOP;
    end loop ;
    DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;
end ;

```

FIG. 1.2.3a

```

with INTRPLEC ; use INTRPLEC ;
with INTRPLIO ; use INTRPLIO ;

```



```

procedure INTRPLAN is
begin
  RETRIEVE_PREVIOUS_INPUTS_FROM_DISK ;
  LET_USER_EDIT_INPUT_DATA ;
  SAVE_EDITED_INPUTS_ON_DISK ;
  SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
  DISPLAY_DATA_SHELL ( NOMINAL_DEPARTURE_DELTA_V ) ;
  for J in 0..10 loop
    COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
    INNER_LOOP;
  end loop ;
  DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;
end ;

```

FIG. 1.2.3b

```

procedure INNER_LOOP is
begin
  for I in 0..16 loop
    COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET ( I ) ;
    COMPUTE_TRAJECTORY_DATA ( I, J ) ;
    DISPLAY_VALUE ( TOTAL_DELTA_V , I, J ) ;
    CHECK_FOR_INTERRUPT_FROM_KEYBOARD ;
  end loop ;
end INNER_LOOP;

```

FIG. 1.2.3c

The module INNER_LOOP is the newly created module in order to incorporate the inner FOR loop in the original main program of both INTRPLAN and IPCAPTUR. As a matter of fact, this new procedure automatically have become a reusable component. Further, the new main program is just a set of module calls with one single loop structure. But the FOR loop must be changed to a WHILE loop as to fulfil ESL requirements. We have discussed this later in this section.

The above modification is inadequate. Of interest to us is whether, the modified main programs INTRPLAN and IPCAPTUR can be represented in an ESL graph. A straight answer is NO. Still we need to change the outer FOR loop structure. We can think of replacing the outer FOR loop structure with a WHILE loop structure as ESL supports WHILE loops. In order to do this, the value of J must be incremented inside the WHILE loop. This can be implemented with a simple computational statement like $J := J + 1$;

```

with IPCAPTEC ; use IPCAPTEC ;
with IPCAPTIO ; use IPCAPTIO ;

procedure IPCAPTUR is

```

```

LOOP_END : boolean := FALSE;
J : integer := 1;
begin
RETRIEVE_PREVIOUS_INPUTS_FROM_DISK ;
LET_USER_EDIT_INPUT_DATA ;
SAVE_EDITED_INPUTS_ON_DISK ;
SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
DISPLAY_DATA_SHELL ( TOTAL_DELTA_V ) ;
while LOOP_END = FALSE loop
    COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
    INNER_LOOP;
    J := J + 1;
    if J > 10 then
        LOOP_END := TRUE;
    end if;
end loop
DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;
end

```

FIG. 1.2.4

The above is the modified main program code for IPCAPTUR. (Considering the main program of IPCAPTUR is good enough for the time being). Changing the inner FOR loop into a WHILE loop caused us to incorporate few other additional statements (FIG. 1.2.5) within the WHILE loop.

```

J := J + 1;
if J > 10 then
    LOOP_END := TRUE;
end if;

```

FIG. 1.2.5

Added Computational Statements inside the WHILE loop

The question is whether the modified main program shown in figure 1.2.4 is good enough to construct an ESL graph. Again, a straight answer is NO. The simple reason is that, there cannot be any computational statements within a piece of code except for a set of module calls , to construct the corresponding ESL representation. Hence a solution is to further decompose the main-program (of INTRPLAN & IPCAPTUR); meaning, removing the outer FOR loop and incorporate it in a separate module, and call that module from the main program. The figure 1.2.6 shows the final picture of the main program for INTRPLAN and IPCAPTUR.

```

with IPCAPTEC ; use IPCAPTEC ;
with IPCAPTIO ; use IPCAPTIO ;

```

```

procedure IPCAPTUR is
begin
RETRIEVE_PREVIOUS_INPUTS_FROM_DISK      ;
LET_USER_EDIT_INPUT_DATA                ;
SAVE_EDITED_INPUTS_ON_DISK              ;
SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
DISPLAY_DATA_SHELL                      ( TOTAL_DELTA_V ) ;
MAIN_LOOP;
DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;
end ;

```

FIG. 1.2.6a

```

with INTRPLEC ; use INTRPLEC ;
with INTRPLIO ; use INTRPLIO ;

procedure INTRPLAN is
begin
RETRIEVE_PREVIOUS_INPUTS_FROM_DISK      ;
LET_USER_EDIT_INPUT_DATA                ;
SAVE_EDITED_INPUTS_ON_DISK              ;
SET_UP_CONSTANTS_AND_PLANETARY_EPHEMERIDES ;
DISPLAY_DATA_SHELL                      ( NOMINAL_DEPARTURE_DELTA_V ) ;
MAIN_LOOP;
DISPLAY_TRAJECTORY_DATA_OF_INTEREST_TO_USER ;
end ;

```

FIG. 1.2.6b

where MAIN_LOOP is the newly created procedure to incorporate the outer loop in the main program(s) (FIG 1.2.7).

```

procedure MAIN_LOOP is
begin
  for J in 1..10 loop
    COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
    INNER_LOOP;
  end loop;
end MAIN_LOOP;

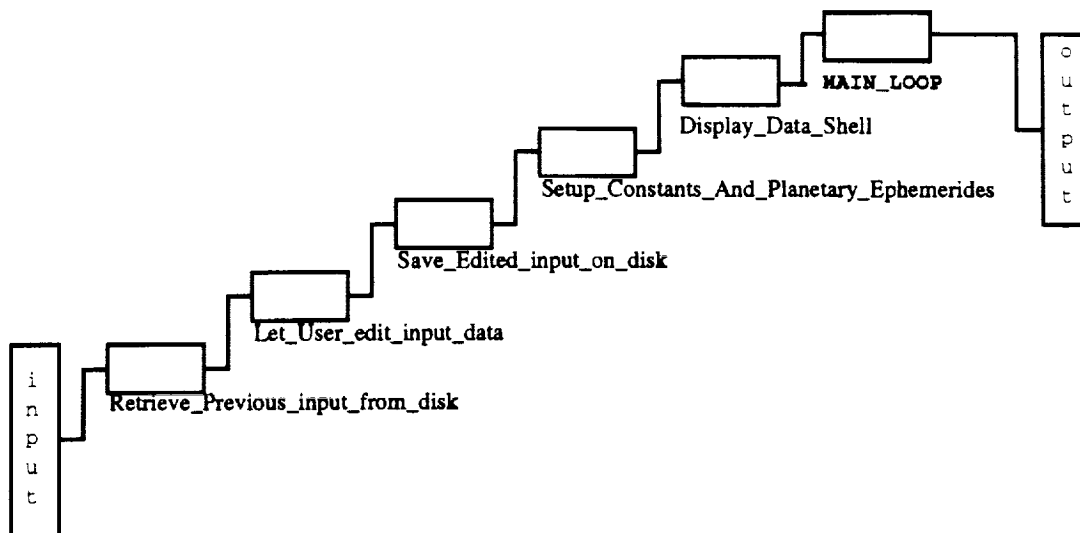
```

FIG. 1.2.7

The final main program(s) is purely a set of module calls and within ESL requirements. The ESL graphical representation to create the main program structure is shown in FIG. 1.2.8.

FIG. 1.2.8

ESL Graphical Representation of The Main Program for INTRPLAN or IPCAPTUR



The decomposition of the main program(s) caused create two new procedures INNER_LOOP and MAIN_LOOP. Obviously, these two procedures have the format of a ESL sub program where, only module calls are allowed. But first we need to modify the module MAIN_LOOP. Introduction of a WHILE loop and to have a separate procedure for the portion shown in fig.1.2.5 would be the main modifications. Fig. 1.2.9 illustrates the MAIN_LOOP after the modifications.

```

procedure MAIN_LOOP is
  LOOP_END : boolean := false;
  CONST : integer CONSTANT := 10 ;

begin
  while LOOP_END = false loop
    COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET ( J ) ;
    INNER_LOOP;
    SET_CONTROL(J, LOOP_END, CONST);
  end loop;
end MAIN_LOOP;
  
```

FIG. 1.2.9

where SET_CONTROL is another new procedure, created to incorporate the small portion of code shown in fig. 1.2.5. This is shown in FIG. 1.2.10

```

procedure SET_CONTROL( J_IN : integer; DONE : boolean; CONST : integer) is
begin
    J_IN := J_IN + 1;
    if J_IN > CONST then
        DONE := TRUE;
    end if;
end SET_CONTROL;

```

FIG. 1.2.10

The benefit of making this modifications is that the software component SET_CONTROL is now converted to a reusable module. Hence this same module can be called by the procedure INNER_LOOP, by making similar modifications as done for the module MAIN_LOOP. Fig. 1.2.11 shows the modified procedure INNER_LOOP.

```

procedure INNER_LOOP is
LOOP_END : boolean := FALSE;
CONST : integer CONSTANT := 16;
begin
    while LOOP_END = FALSE loop
        COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET ( I      )      ;
        COMPUTE_TRAJECTORY_DATA                      ( I, J )      ;
        DISPLAY_VALUE                                ( TOTAL_DELTA_V , I, J )      ;
        CHECK_FOR_INTERRUPT_FROM_KEYBOARD              ;
        SET_CONTROL(J, LOOP_END, CONST);
    end loop;
end INNER_LOOP;

```

FIG 1.2.11

It is now very clear that the two procedures INNER_LOOP and the MAIN_LOOP are converted into ESL subprograms. Figures 1.2.12 and 1.2.13 illustrate the ESL graphical representation of the two subprograms.

FIG. 1.2.12

The ESL object graph for subprogram MAIN_LOOP

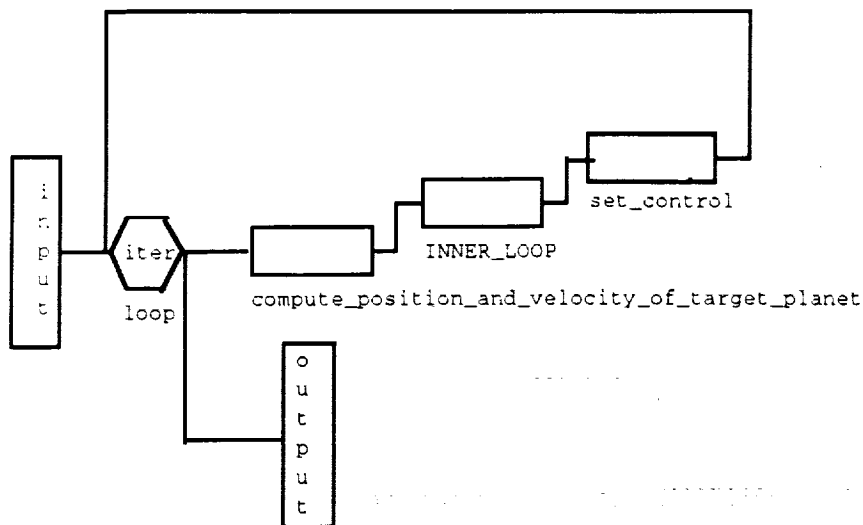
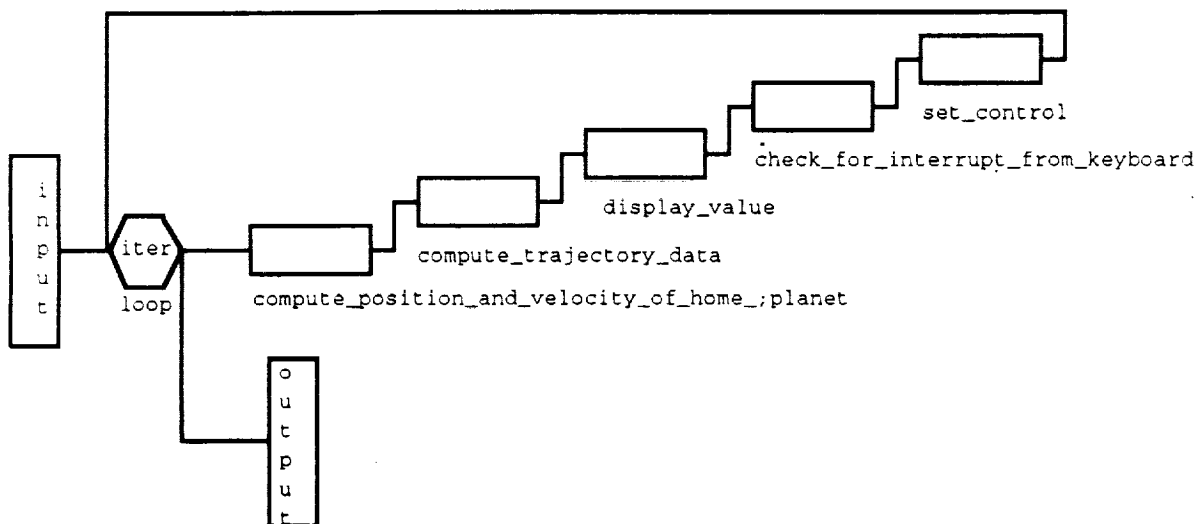


FIG. 1.2.13

The ESL object graph for subprogram INNER_LOOP



The same ESL object graph could be used for BEST1WAY . It is important to make sure that the user set proper constant values when modules being called for individual applications. For example, the constant value passed into the reusable module SET_CONTROL, must be properly set inside procedures MAIN_LOOP and INNER_LOOP. i.e values 10 and 16 respectively for INTRPLAN and IPCAPTUR. Similarly, for BEST1WAY.

Comparatively, main program for POWERSWNG looks slightly different to the main programs of the other three applications. But of course, many of the modules already modified for reusable purposes can be used in designing ESL object graph for POWERSWNG. For POWERSWNG, the following procedure calls, must be added.

COMPUTE_POSITION_AND_VELOCITY_OF_SWINGBY_PLANET
 COMPUTE_TRAJECTORY_FOR_FIRST_HELIOCENTRIC_LEG
 DISPLAY_VALUE1
 DISPLAY_VALUE2
 COMPUTE_TRAJECTORY_FOR_SECOND_HELIOCENTRIC_LEG

The following is the ESL graphical representation for POWERSWNG.

FIG. 1.2.14

ESL Object graph for main program of POWERSWNG

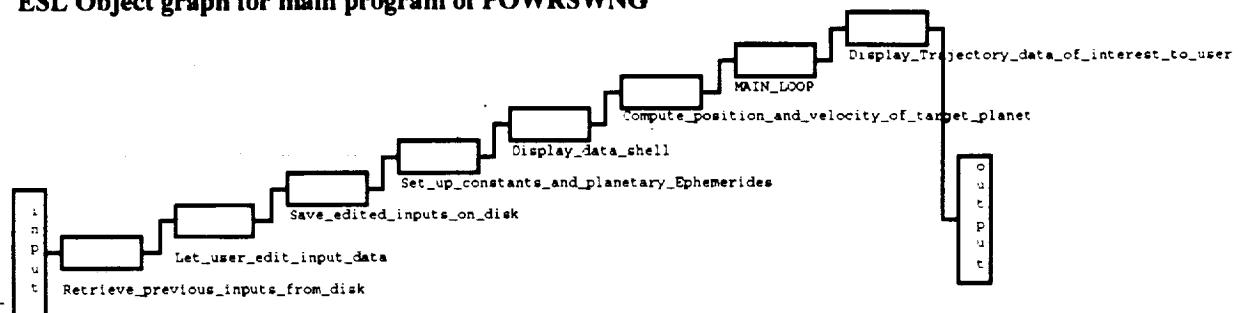


FIG. 1.2.15

The ESL object graph for subprogram MAIN_LOOP

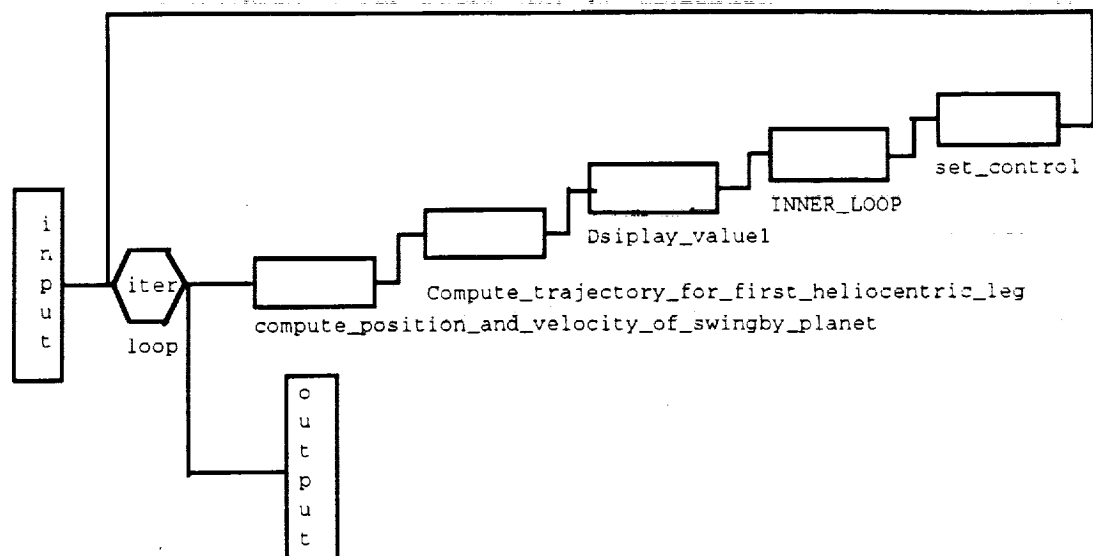
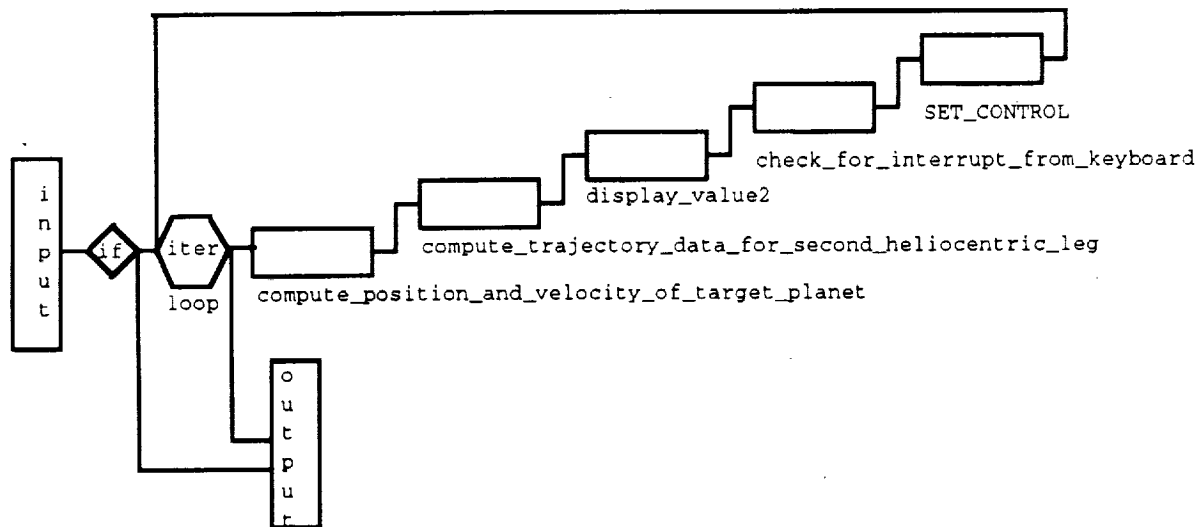


FIG. 1.2.16

The ESL object graph for subprogram INNER_LOOP of POWRSWNG



1.2.2 Modifications and Decomposition of Primitives.

In ESL terms, Primitives are the modules that cannot be further decomposed or modules that are not worth decomposing. For example the module `RETRIVE_PREVIOUS_INPUTS_FROM_DISK` is a repeated module in all four applications in question. Though the module in `POWRSWNG` is slightly different to the module in other three applications, all four modules serve the same purpose. Further decomposition is out of question. Hence, best option is to have a single module that serves all four applications, making that a reusable component. Of course, to build a common reusable module, modifications are need to be carried out.

Let us look into the modifications that have been done in order to make this module a reusable component. Originally, not a single parameter was passed into the procedure. As a major modification, two new parameters have been introduced namely `FILE_NAME` of type `string` and `NAME_IN` of type `APPLICATION_TYPE`. `APPLICATION_TYPE` is a user defined type and initially has the enumerated type values `INTRPLAN`, `IPCAPTURE`, `BEST1WAY`, and `POWRSWNG`. `NAME_IN` passes in the appropriate value based on the application. `FILE_NAME` is the data_file name relevant to each application. In other words the corresponding data_file name for `INTRPLAN` is `intrplan.get`. Similarly others. Inside the module, `CASE` and `IF_THEN_ELSE` structures have been introduced to serve different application types.

Modifications have been made to the following procedures in a similar manner.

```
LET_USER_EDIT_INPUT_DATA  
SAVE_EDITED_INPUT_ON_DISK  
KILL_OUTDATED_INPUT_FILE  
DISPLAY_LINE_LEADERS  
DISPLAY_FOOTER_LINES
```

In each one of the above modules, a new input parameter of `APPLICATION_TYPE` is introduced. This parameter passes the name of the application that uses this module into the module. This helps to serve the needs of each application program. For instance, `POWRSWNG` performs a slightly different task in many of the above modules. Passing in the name of the application helps direct the

execution to the specific area within the module where those different tasks are carried out.

1.2.3. Packages COMMON_MODULES, DATA_TYPES, DATA_TYPES_SPEC

The packages COMMON_MODULES, DATA_TYPES, DATA_TYPES_SPEC COMMON_MODULES are the three new packages introduced into the system. The services provided by these packages are described below.

Package COMMON_MODULES.

This is a newly created package build to include all the common reusable procedures and functions. Also this package includes newly created reusable modules as a result of decomposition. For instance, the modules described in section 1.2.1 namely MAIN_LOOP, INNER_LOOP and SET_CONTROL, are residing in package COMMON_MODULES. Of course there are many more modules residing in this package, which we will be discussing later in this report.

Package DATA_TYPES.

This is also a newly created package to include all the type declarations and variable declarations, which are also repeated in all four applications. However this package includes only the data types and type declarations that are found in package bodies of all four application programs.

Package DATA_TYPES_SPEC.

This package is similar to the package DATA_TYPES. This package is created to include all the data types defined in the specifications of application programs.

It is important to make a note that packages DATA_TYPES and DATA_TYPES_SPEC are now directly reusable as all the application programs use these packages.

1.2.4 Further Modifications.

After a thorough analysis of the modules

1. COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET,
2. COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET,
3. COMPUTE_POSITION_AND_VELOCITY_OF_SWINGBY_PLANET,

it was found that these modules are very similar and perform the same task. Therefore, it is obvious that, from these three modules a single reusable module can be built.

procedure COMPUTE_POSITION_AND_VELOCITY_OF_HOME_PLANET (I : integer) is

```

    DT_SECS : FLOTE ;

begin
    JDATE(HOME) := NOM_JDATE(HOME) + LONG_FLOTE(I-8) * INTERVAL(HOME) ;
    DT_SECS := 86400.0 * FLOTE( JDATE(HOME) - PER_JDATE(HOME) ) ;
    PROPAGATE_POSITION_AND_VELOCITY_THRU_TIME (
        PER_HELIPOS(HOME) , PER_HELIVEL(HOME) , DT_SECS , GM_SUNp(HOME) ,
        HELIPOS(HOME) , HELIVEL(HOME)
    ) ;
end

```

FIG. 1.2.17

procedure COMPUTE_POSITION_AND_VELOCITY_OF_TARGET_PLANET (J : integer) is

```

    DT_SECS : FLOTE ;

begin
    JDATE(TARG) := NOM_JDATE(TARG) + LONG_FLOTE(J-5) * INTERVAL(TARG) ;
    DT_SECS := 86400.0 * FLOTE( JDATE(TARG) - PER_JDATE(TARG) ) ;
    PROPAGATE_POSITION_AND_VELOCITY_THRU_TIME (
        PER_HELIPOS(TARG) , PER_HELIVEL(TARG) , DT_SECS , GM_SUNp(TARG) ,
        HELIPOS(TARG) , HELIVEL(TARG)
    ) ;
end

```

FIG. 1.2.18

procedure COMPUTE_POSITION_AND_VELOCITY_OF_SWINGBY_PLANET (J : integer) is

```

    DT_SECS : FLOTE ;

begin
    JDATE(SWBY) := NOM_JDATE(SWBY) + LONG_FLOTE(J-5) * INTERVAL(SWBY) ;
    DT_SECS := 86400.0 * FLOTE( JDATE(SWBY) - PER_JDATE(SWBY) ) ;
    PROPAGATE_POSITION_AND_VELOCITY_THRU_TIME (
        PER_HELIPOS(SWBY) , PER_HELIVEL(SWBY) , DT_SECS , GM_SUNp(SWBY) ,
        HELIPOS(SWBY) , HELIVEL(SWBY) ) ;
end
;
```

FIG. 1.2.19

Shown above are the three procedures found in all four applications. As we have said earlier, simply these modules do the same task except for a few minor differences. The module shown below is a procedure built in order to perform all three tasks, and is reusable.

```

procedure COMPUTE_POSITION_AND_VELOCITY_OF_PLANET ( I : integer;
                                                    PLANET : PLANET_TYPE;
                                                    NAME : APPLICATION_TYPE ) is

    DT_SECS : FLOTE ;
    TEMP : TRAG_NODE ;
    COUNTER : integer;

begin
    case PLANET is
        when TARGET | target => TEMP := TRAG;
            if NAME = POWRSWNG then
                COUNTER := I - 6;
            else
                COUNTER := I - 5;
            end if;
        when HOME | home      => TEMP := HOME;
            COUNTER := I - 8;
        when SWNGBY | swngby => TEMP := SWBY;
            COUNTER := I - 5;
        when others          => null;
    end case;
    if NAME = POWRSWNG AND DESTINATION = SWNGBY then
        JDATE(TEMP) := NOM_JDATE(TEMP);
    else
        JDATE(TEMP) := NOM_JDATE(TEMP) + LONG_FLOTE(COUNTER) * INTERVAL(TEMP) ;
    end if;
    DT_SECS := 86400.0 * FLOTE( JDATE(TEMP) - PER_JDATE(TEMP) ) ;
    PROPAGATE_POSITION_AND_VELOCITY_THRU_TIME (
        PER_HELIPOS(TEMP) , PER_HELIVEL(TEMP) , DT_SECS , GM_SUNp(TEMP) ,

```

```

        HELIPOS(TEMP) ,      HELIVEL(TEMP)
end  COMPUTE_POSITION_AND_VELOCITY_OF_PLANET;

```

FIG. 1.2.20

This new procedure is named **COMPUTE_POSITION_AND_VELOCITY_OF_PLANET**, and have three new parameters namely **I** of type integer, **DESTINATION** of type **DESTINATION_TYPE** and **NAME_IN** of type **APPLIATION_TYPE**. **DESTINATION_TYPE** is also a user defined type and it defines the destination (**HOME**, **TARGET** or **SWINGBY**). **APPLICATION_TYPE** is the same type described earlier.

At this point, it is important to make a note that, creating a new module by the name **COMPUTE_POSITION_AND_VELOCITY_OF_PLANET** will change the corresponding object name in ESL object graph shown in section 1.2.2

FIG. 1.2.21

The ESL object graph for subprogram **MAIN_LOOP**

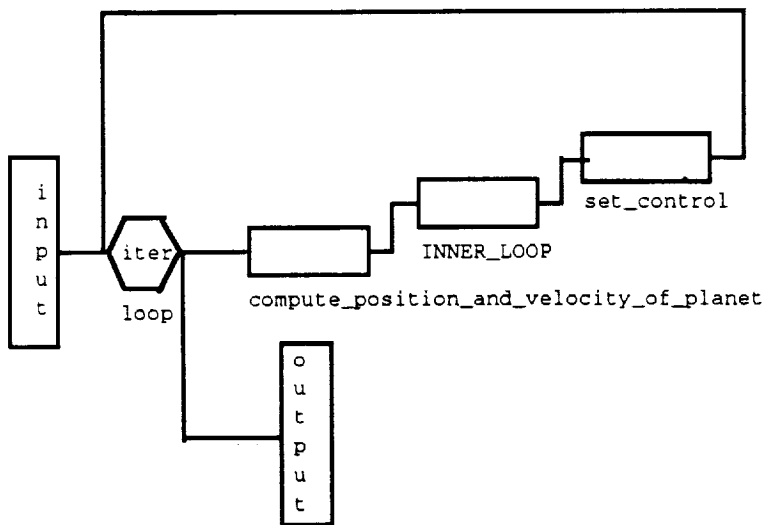
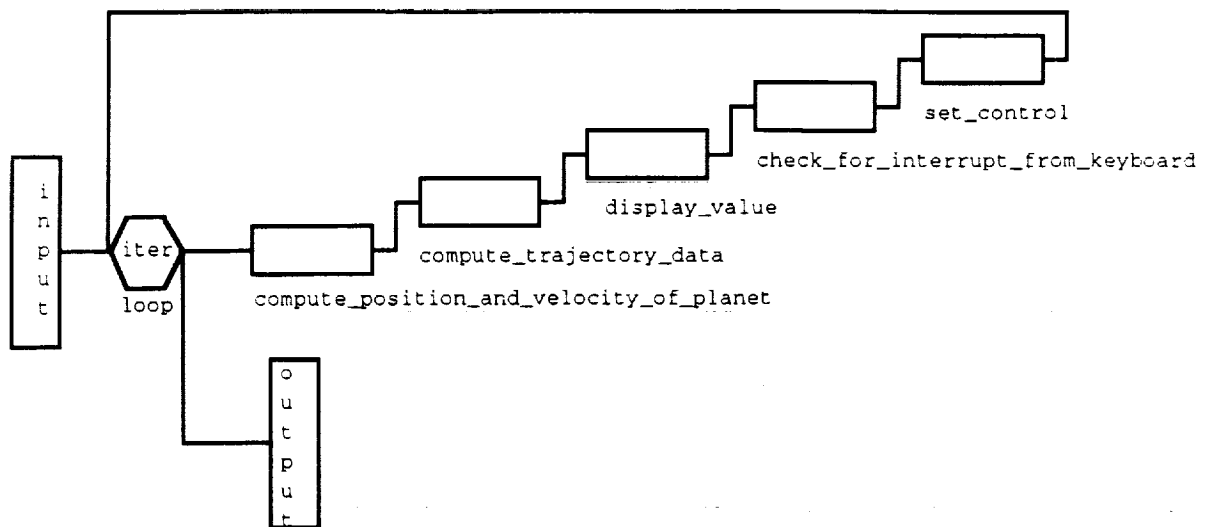


FIG. 1.2.22

The ESL object graph for subprogram INNER_LOOP



Similarly the object names

compute_position_and_velocity_of_home_planet

compute_position_and_velocity_of_target_planet

compute_position_and_velocity_of_swingby_planet

in figures 1.2.15, 1.2.16 and 1.2.18 for POWRSWNG will change accordingly.

1.2.5 Modification of procedure COMPUTE_TRAJECTORY_DATA

Compute_trajectory_data is a another procedure available in all four applications. A thorough analysis revealed that this procedure is an ideal module to decompose and convert into a ESL sub program. Decomposition had to be done so that the grains (decomposed components) could be reused in other similar modules throughout the applications. One major change made in reengineering this module is to eliminate exception handlers. May be this looks very inappropriate, but elimination of exception handlers was necessary to convert this module into a ESL sub program. We know that in ESL a sub program allows only a set of procedure or function calls . Also we need to realize that all these changes must be done having ESL in mind. At this point we need to think of how to tackle the granularity problem. i.e how big a grain is ?. The reason is that, when

decomposing the module, very small grains of size one, two or three lines remains within the module. In ESL terms, we cannot leave them within a module. We are forced to eliminate them and reside them in separate modules.

Let us take a look at how decomposition was done. FIG. 1.2.23 shows decomposed grains by drawing lines in between. Each grain is residing in a procedure with a appropriate procedure name.

```

procedure COMPUTE_TRAJECTORY_DATA ( I, J : integer ) is

    TOO_FAST      : exception ;
    TOO_HOT       : exception ;

    SINFAC        : FLOTE ;
    TEST_VEC      : VECTOR ;
    TF_DAYS       : FLOTE ;

begin
    -----EXCEPTION_HANDLER_10001-----
    TF_DAYS := FLOTE( JDATE(TARG) - JDATE(HOME) ) ;
    if abs( TF_DAYS ) <= 20.0 then
        raise TOO_FAST
    end if
    -----CALCULATIONS-----
    if TF_DAYS > 0.0
    then DEP := HOME
    else DEP := TARG
    end if
    if DEP = HOME
    then ARR := TARG
    else ARR := HOME
    end if
    TF_SECS := 86400.0 * abs( TF_DAYS )
    ANGMO_PREF := HELIPOS(DEP) * HELIVEL(DEP)
    TEST_VEC := HELIPOS(DEP) * HELIPOS(ARR) ; -- * gives cross product
    if TEST_VEC & ANGMO_PREF < 0.0 -- & gives dot product
    then SINFAC := -ABS( TEST_VEC )
    else SINFAC := +ABS( TEST_VEC )
    end if
    XFR_ANG := FULL_REVS*TWOPI + ATAN1( SINFAC, HELIPOS(DEP)&HELIPOS(ARR) ) ;
    DVALUE( HELIOCENTRIC_TRANSFER_ANGLE )(I,J) := DATA_MATRIX_INTEGER(
        DEGPERRAD * XFR_ANG ) ;
    DVALUE( FLIGHT_TIME )(I,J) := DATA_MATRIX_INTEGER( TF_DAYS )
    -----
    FIND_BEST_TRANSFER_TRAJECTORY
    -----CALL_FOR_EXCEPTION_HANDLER_10003_10004
    if ( FULL_REVS > 0 ) and ( SMA_SIZE /= BEST_SIZE ) then
        SOLVE_LAMBERT_PROBLEM ( HELIPOS(DEP), TF_SECS, HELIPOS(ARR),
            ANGMO_PREF, XFR_HELIVEL(DEP), XFR_HELIVEL(ARR),
            GM_SUN, FULL_REVS, BEST_SIZE )
    end if
    -----EXCEPTION_HANDLER_10002-----
    if ARRIVAL_SPEED_PENALTY > 0.0 then
        raise TOO_HOT
    end if
    -----

```

```

COMPUTE_HELIOCENTRIC_TRAJECTORY_DATA ( I, J )      ;
COMPUTE_PLANETOCENTRIC_DEPARTURE_DATA ( I, J )      ;
COMPUTE_PLANETOCENTRIC_ARRIVAL_DATA ( I, J )      ;

pragma page ;

exception
when TOO_FAST =>
    for KIND in DATA_KIND loop
        DVALUE( KIND )(I,J) := 10001
    end loop
when TOO_HOT =>
    for KIND in MULTIREV_SEMIMAJOR_AXIS..APHELION_DISTANCE loop
        DVALUE( KIND )(I,J) := 10002
    end loop
when LAMBERT_Z_ITERATION_FAILED_TO_CONVERGE =>
    for KIND in MULTIREV_SEMIMAJOR_AXIS..APHELION_DISTANCE loop
        DVALUE( KIND )(I,J) := 10003
    end loop
when LAMBERT_CANNOT_ATTAIN_SPECIFIED_NUMBER_OF_REVS =>
    for KIND in MULTIREV_SEMIMAJOR_AXIS..APHELION_DISTANCE loop
        DVALUE( KIND )(I,J) := 10004
    end loop

end

```

FIG. 1.2.23

All exceptions are handled within the same module where the exception is raised. For example, consider the newly created procedure `EXCEPTION_HANDLER_10001`. The exception is raised if the absolute value of `TF_DAYS` is less than or equal to 20.0. The module is reengineered in such a way that the sequence of instructions that are to be executed the moment this exception is raised are within the same procedure itself. This is illustrated in FIG. 1.2.24.

```

Procedure EXCEPTION_HANDLER_10001(TF_DAYS : flote; DONE : boolean; NAME : in
APPLICATION_TYPE; CATEGORY : CATEGORY_TYPE) IS

Begin
    if NAME = POWRSWNG then
        if CATEGORY = LEG1 then
            TF_days := flote(Jdate(SWBY) - JDATE(HOME));
            if abs(TF_DAYS) <= 20.0 then
                for kind1 in LEG1_HELIOCENTRIC_TRANSFER_ANGLE..LEG1_FLIGHT_TIME loop
                    VALUE1(KIND1)(J) := 10001;
                end loop;
            end if;
            DONE := true;
        elsif CATEGORY = LEG2 then
            TF_days := flote(Jdate(TRAJ) - JDATE(SWBY));
            if abs(TF_DAYS) <= 20.0 then

```



```

        for kind1 in LEG2_HELIOCENTRIC_TRANSFER_ANGLE..LEG2_FLIGHT_TIME loop
            VALUE2(KIND2)(I,J) := 10001;
        end loop;
    end if;
    DONE := true;
end if;

else
    TF_days := flote(Jdate(TRAG) - JDATE(HOME));
    if abs(TF_DAYS) <= 20.0 then
        for kind in DATA_KIND loop
            DVALUE(KIND)(I,J) := 10001;
        end loop;
        DONE := true;
    end if;
end if;
end EXCEPTION_HANDLER_10001;

```

FIG. 1.2.24

The variable **TF_DAYS** should be passed-in from the module **COMPUTE_TRAJECTORY_DATA** because it is declared inside that module. Moreover, three new parameters **NAME_IN**, **CATEGORY** and a boolean variable **DONE** are passed into the module. **CATEGORY** is of user defined type **CATEGORY_TYPE** and have elements (**LEG1** and **LEG2**).

In the original code of this module, once the exception is raised, the execution is passed to the area where the exception is defined. Once that area is executed, the control will transferred to the end of the module. In the reengineered module, this is handled by a if-then-else structure. We have selected to introduce an if-then-else structure because **ESL** supports such structures. Hence the reengineered procedure **COMPUTE_TAJECTORY_DATA** will have the following format and is a sub program within **ESL** requirements.

```

procedure COMPUTE_TRAJECTORY_DATA ( I, J : integer ) is

    SINFAC      : FLOTE ;
    TEST_VEC     : VECTOR ;
    TF_DAYS      : FLOTE ;
    DONE_1, DONE_2, DONE_3 : boolean := false;
    DESTINATION_D : DESTINATION_TYPE := DEPARTURE;
    DESTINATION_A : DESTINATION_TYPE:= ARRIVAL;
    NAME : APPLICATION_TYPE:= INTRPLEC;
    CATEGORY := DUMMY;

begin
    EXCEPTION_HANDLER_10001(TF_DAYS,DONE_1, NAME , CATEGORY);
    if DONE_1 = false then
        CALCULATIONS(TF_DAYS, SINFAC, TEST_VEC);
        FIND_BEST_TRANSFER_TRAJECTORY;
        CALL_FOR_EXCEPTION_HANDLER_10003_10004(DONE_2,CATEGORY,NAME);
    end if;
end;

```

```

if DONE_2 = false then
  EXCEPTION_HANDLER_10002(DONE_3, NAME);
  if DONE_3 = true then
    COMPUTE_HELIOCENTRIC_TRAJECTORY_DATA( I, J, NAME );
    COMPUTE_PLANETOCENTRIC_ARRIVAL_OR_DEPARTURE_DATA(I, J, DESTINATION_D, NAME);
    COMPUTE_PLANETOCENTRIC_ARRIVAL_OR_DEPARTURE_DATA(I, J, DESTINATION_A, NAME )
  end if;
end if;
end if;
end COMPUTE_TRAJECTORY_DATA;

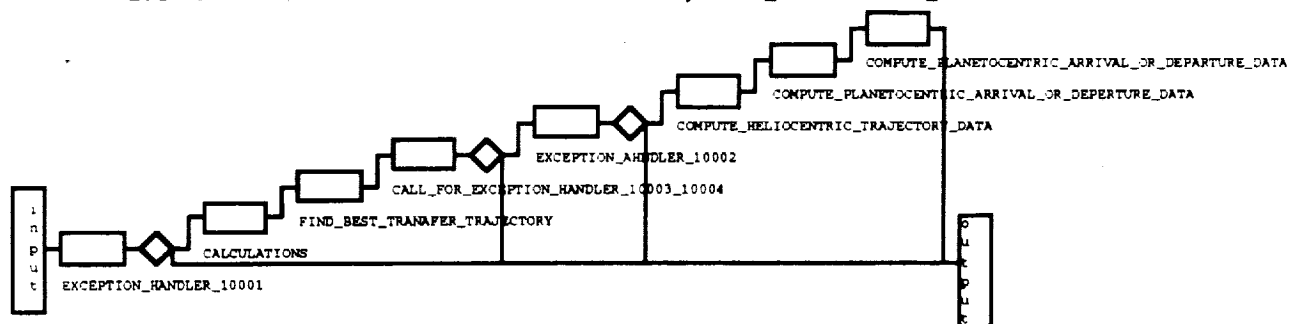
```

FIG.1.2.25

The corresponding ESL object graph diagram for the above sub program is shown in fig. 1.2.26.

FIG. 1.2.26

ESL OBJECT DIAGRAM FOR THE SUB PROGRAM COMPUTE_TRAJECTORY_DATA



The ESL object graphs of the same sub program in BEST1WAY and POWRSWNG are slightly different to the above. The graph shown above is the ESL object graph for INTRPLAN. The ESL object graph for IPCAPTUR is almost the same except for less one procedure call(i.e exception_handler_10002).

In the original program code for the procedure COMPUTE_TRAJECTORY_DATA, we see two procedure calls by the names COMPUTE_PLANETOCENTRIC_DEPARTURE_DATA and COMPUTE_PLANETOCENTRIC_ARRIVAL_DATA. Since the two procedures do the same task, we could have one procedure to handle both situations and building another reusable module.

```

procedure COMPUTE_PLANETOCENTRIC_DEPARTURE_DATA ( I,J : integer ) is

    DECL : integer ;
    DELV : FLOTE ;
    DV : integer ;
    RASC : integer ;
    VINHAT : VECTOR ;
    VINMAG : FLOTE ;
    VINVEC : VECTOR ;
    VINF : integer ;

begin
    VINVEC := (XFR_HELIVEL(DEP)-HELIVEL(DEP))*VBASE_M50_E(DEP) ;
    VINMAG := ABS( VINVEC ) ;
    VINHAT := VINVEC / VINMAG ;
    DELV := DEPARTURE_VELOCITY_INCREMENT ;
    DV := DATA_MATRIX_INTEGER( DELV * 100 ) ;
    VINF := DATA_MATRIX_INTEGER( VINMAG * 100 ) ;
    DECL := ROUND( ASIN( VINHAT(3) ) * 1800 / PI ) ;
    RASC := ROUND( ATAN1( VINHAT(2) , VINHAT(1) ) * 1800 / PI ) ;
    DVALUE( NOMINAL_DEPARTURE_DELTA_V )(I,J) := DV ; -- dkm/sec
    DVALUE( DEPARTURE_V_INFINITY_MAGNITUDE )(I,J) := VINF ; -- dkm/sec
    DVALUE( DEPARTURE_V_INFINITY_DECLINATION )(I,J) := DECL ; -- 0.1 deg
    DVALUE( DEPARTURE_V_INFINITY_RTASCENSION )(I,J) := RASC ; -- 0.1 deg
end ;

```

FIG. 1.2.27

```

procedure COMPUTE_PLANETOCENTRIC_ARRIVAL_DATA ( I,J : integer ) is

    DECL : integer ;
    RASC : integer ;
    VINHAT : VECTOR ;
    VINMAG : FLOTE ;
    VINVEC : VECTOR ;
    VINF : integer ;
    SPEED : FLOTE ;
    SPD : integer ;

begin
    VINVEC := (XFR_HELIVEL(ARR)-HELIVEL(ARR))*VBASE_M50_E(ARR) ;
    VINMAG := ABS( VINVEC ) ;
    VINHAT := VINVEC / VINMAG ;

```

```

SPEED := SQRT( VESQ(ARR) + VINMAG*VINMAG )
SPD   := DATA_MATRIX_INTEGER(      SPEED      * 100      ) ;
VINF   := DATA_MATRIX_INTEGER(      VINMAG     * 100      ) ;
DECL   := ROUND(      ASIN ( VINHAT(3) ) * 1800 / PI ) ;
RASC   := ROUND(      ATAN1( VINHAT(2) , VINHAT(1) ) * 1800 / PI ) ;
DVALUE( ARRIVAL_SPEED      )(I,J) := SPD      ; -- dkm/sec
DVALUE( ARRIVAL_V_INFINITY_MAGNITUDE )(I,J) := VINF      ; -- dkm/sec
DVALUE( ARRIVAL_V_INFINITY_DECLINATION )(I,J) := DECL      ; -- 0.1 deg
DVALUE( ARRIVAL_V_INFINITY_RTASCENSION )(I,J) := RASC      ; -- 0.1 deg
end

```

FIG. 1.2.28

```

procedure COMPUTE_PLANETOCENTRIC_ARRIVAL_OR_DEPARTURE_DATA ( I,J: integer;
                                                             DESTINATION : DESTINATION_TYPE;
                                                             NAME: APPLICATION_TYPE) is

DECL : integer;
RASC : integer;
VINHAT: VECTOR;
VINMAG: FLOTE;
VINVEC: VECTOR;
VINF : integer;
DV_OR_SPD : integer;
TEMP: TRAJ_NODE;
NDDV_OR_AS : ;
DVIM_OR_AVIM : ;
DVID_OR_AVID : ;
DVIR_OR_AVIR;
TOT_DELV: flote;
TOT_DV : integer;

begin
  case DESTINATION is
    when DEPARTURE | departure => TEMP := DEP;
    when ARRIVAL | arrival    => TEMP := ARR;
  end case;

  VINVEC := (XFR HELIVEL(TEMP) - HELIVEL(TEMP))*VBASE M50 E(TEMP);
  VINMAG := ABS( VINVEC );
  VINHAT := VINVEC / VINMAG;
  if DESTINATION = DEPARTURE then
    DELV_OR_SPEED := DEPARTURE_VELOCITY_INCREMENT;
    if NAME = IPCAPTURE then
      TOT_DELV := ARRIVAL_VELOCITY_INCREMENT + DELV_OR_SPEED;
      TOT_DV := DATA_MATRIX_INTEGER(TOT_DEV * 100);
    end if;

    NDDV_OR_AS := NOMINAL_DEPARTURE_DELTA_V;
    DVIM_OR_AVIM := DEPARTURE_V_INFINITY_MAGNITUDE;
    DVID_OR_AVID := DEPARTURE_V_INFINITY_DECLINATION;
    DVIR_OR_AVIR := DEPARTURE_V_INFINITY_RTASCENSION;
  elsif DESTINATION = ARRIVAL then
    DELV_OR_SPEED := SQRT(VESQ(TEMP) + VINMEG*V-MEG);
    DELV_OR_SPEED := MIN ( DELV_OR_SPEED , MAXAVELMAG(TEMP);
    NDDV_OR_AS := ARRIVAL_SPEED;
    DVIM_OR_AVIM := ARRIVAL_V_INFINITY_MAGNITUDE;
    DVID_OR_AVID := ARRIVAL_V_INFINITY_DECLINATION;
    DVIR_OR_AVIR := ARRIVAL_V_INFINITY_RTASCENSION;
  end if;

  DV_OR_SPD := DATA_MATRIX_INTEGER( DELV_OR_SPEED * 100);
  VINF := DATA_MATRIX_INTEGER(VINMAG * 100);
  DECL := ROUND(ASIN ( VINHAT(3) ) * 1800/ PI ) ;

```

```

RASC := ROUND( ATANI( VINHAT(2), VINHAT(1) ) * 1800 / PI );
if NAME := IPCAPTURE then
  DVALUE( NDDV_OR_AS )(I,J) := DV_OR_SPD ; -- dkrn/sec
  DVALUE( NDDV_OR_AS )(I,J) := DV_OR_SPD ; -- dkrn/sec
  DVALUE( DVIM_OR_AVIM )(I,J) := VINP ; -- dkrn/sec
  DVALUE( DVID_OR_AVID )(I,J) := DECL ; -- 0.1 deg
  DVALUE( DVIR_OR_AVIR )(I,J) := RASC ; -- 0.1 deg
end if
end;

```

FIG. 1.2.29

The figures 1.2.27 and 1.2.28 show the two procedures in question. Fig. 1.2.29 is the modified procedure built to represent both the procedures shown in figures 1.2.27 and 1.2.28. This procedure replaces 8 modules in all four applications. And hence, it is reusable. In order to make this a reusable module, new variables have been introduced along with the necessary modifications. This procedure also resides in the package `common_modules` which is designed to reside all the newly created, and modified modules.

Procedure `COMPUTE_TRAJECTORY_FOR_FIRST_HELIOCENTRIC_LEG`, and `COMPUTE_TRAJECTORY_FOR_SECOND_HELIOCENTRIC_LEG` are available only in applications `BEST1WAY` and `POWRSWNG`. However the procedure in `POWRSWNG` is very similar to the procedure `COMPUTE_TRAJECTORY_DATA` in `INTRPLAN & IPCAPTURE`. Hence this procedure in `POWRSWNG` can be replaced by already designed reusable components and made a separate ESL object graph. But as we have done in earlier cases, these two procedures in `BEST1WAY` have been modified and built one single reusable procedure named `COMPUTE_TRAJECTORY_FOR_FIRST_AND_SECOND_HELIOCENTRIC_LEG`. The following figures show the modifications.

procedure `COMPUTE_TRAJECTORY_FOR_FIRST_HELIOCENTRIC_LEG` is

```

      ANGMO_PREF : VECTOR ;
      TF1_DAYS   : FLOTE ;
      TF1_SECS   : FLOTE ;

begin
  TF1_DAYS := FLOTE( JDATE(SWBY) - JDATE(DEP) ) ;
  TF1_SECS := 86400.0 * TF1_DAYS ;
  ANGMO_PREF := HELIPOS(DEP) * HELIVEL(DEP) ;
  SOLVE_LAMBERT_PROBLEM ( HELIPOS(DEP), TF1_SECS, HELIPOS(SWBY),
ANGMO_PREF,
      XFR_HELIVEL(DEP), XFR_HELIVEL(SWBY), GM_SUN );
  ANTE_SWBY_VINVEC := XFR_HELIVEL(SWBY) - HELIVEL(SWBY) ;

```

end

FIG. 1.2.30

procedure COMPUTE_TRAJECTORY_FOR_SECOND_HELIOCENTRIC_LEG is

```
    ANGMO_PREF : VECTOR ;
    TF2_DAYS   : FLOTE ;
    TF2_SECS   : FLOTE ;

begin
    TF2_DAYS := FLOTE( JDATE(ARR) - JDATE(SWBY) ) ;
    TF2_SECS := 86400.0 * TF2_DAYS ;
    ANGMO_PREF := HELIPOS(ARR) * HELIVEL(ARR) ;
    SOLVE_LAMBERT_PROBLEM ( HELIPOS(SWBY), TF2_SECS, HELIPOS(ARR),
    ANGMO_PREF,
    XFR_HELIVEL(SWBY), XFR_HELIVEL(ARR), GM_SUN );
    POST_SWBY_VINVEC := XFR_HELIVEL(SWBY) - HELIVEL(SWBY) ;
end
```

FIG. 1.2.31

The following is the modified version of the above two modules.

```
procedure COMPUTE_TRAJECTORY_FOR_FIRST_AND_SECOND_HELIOCENTRIC_LEG(CATEGORY :
                                                                    CATEGORY_TYPE) is

-- THIS IS A REUSABLE COMMON MODULE FOR COMPUTE_TRAJECTORY_FOR_FIRST_HELIOCENTRIC_LEG
--& COMPUTE_TRAJECTORY_FOR_SECOND_HELIOCENTRIC_LEG --
-- CHANGES HAVE BEEN MADE ACCORDINGLY.

    ANGMO_PREF : VECTOR ;
    TF1_DAYS   : FLOTE ;

begin
    if CATEGORY = LEG1 then
        TF1_DAYS := FLOTE( JDATE(SWBY) - JDATE(DEP) ) ;
        ANGMO_PREF := HELIPOS(DEP) * HELIVEL(DEP) ;

        SOLVE_LAMBERT_PROBLEM ( HELIPOS(DEP), TF1_DAYS* 86400.0, HELIPOS(SWBY),
        ANGMO_PREF, XFR_HELIVEL(DEP), XFR_HELIVEL(SWBY), GM_SUN );
        ANTE_SWBY_VINVEC := XFR_HELIVEL(SWBY) - HELIVEL(SWBY) ;

    else
        TF1_DAYS := FLOTE( JDATE(ARR) - JDATE(SWBY) ) ;
        ANGMO_PREF := HELIPOS(ARR) * HELIVEL(ARR) ;

        SOLVE_LAMBERT_PROBLEM ( HELIPOS(SWBY), TF1_DAYS * 86400.0 , HELIPOS(ARR),
        ANGMO_PREF, XFR_HELIVEL(SWBY), XFR_HELIVEL(ARR), GM_SUN );
        POST_SWBY_VINVEC := XFR_HELIVEL(SWBY) - HELIVEL(SWBY) ;
    end if;
end
```

1.2.6 Application BEST1WAY

A few modules in BEST1WAY do not match with any of the modules in either POWRSWNG or INTRPLAN or IPCAPTUR. These modules cannot be made reusable because they are unique only to BEST1WAY.

Therefore the following modules remain same in the application BEST1WAY.

FIND_BEST_DIRECT_TRANSFER_TRAJECTORY
FIND_BEST_SWING_BY_TRAJECTORY
ISOLATE_UNPOWERED_SWINGBY_SOLUTION
COMPUTE_PLANETOCENTRIC_SWINGBY_TRAJECTORY_DATA

1.2.7 Application POWRSWNG

As in BEST1WAY, POWRSWNG also has a few unique modules that are used nowhere else. Since these modules cannot be made reusable, they remain unchanged within the application itself. Hence the modules

COMPUTE_LEG1_PLANETOCENTRIC_TRAJECTORY_DATA
COMPUTE_LEG2_PLANETOCENTRIC_TRAJECTORY_DATA
COMPUTE_PLANETOCENTRIC_SWINGBY_TRAJECTORY_DATA

remain unchanged.

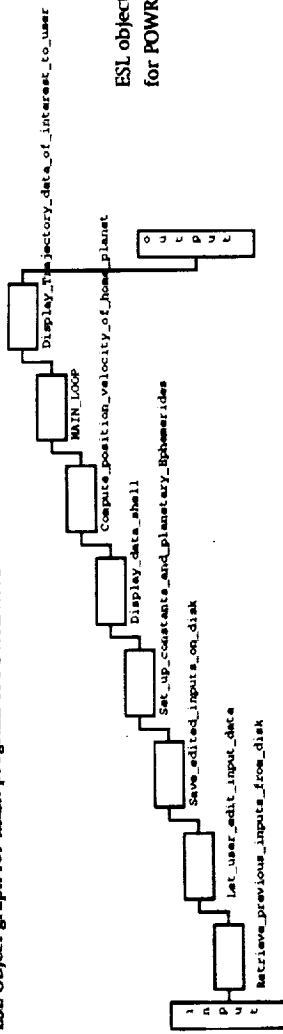
1.2.8 Module DATA_MATRIX_INTEGER

Module DATA_MATRIX_INTEGER is one common module found in all four applications. This module can be used in all applications without any modifications. Therefore this module has been shifted into the package COMMON_MODULES where all the reusable common modules reside.

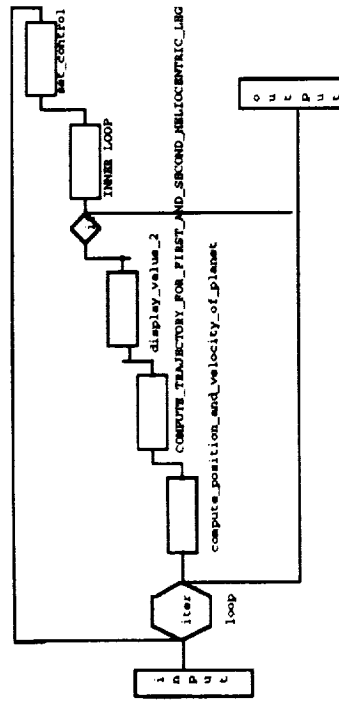
1.2.9 Complete ESL object graphs for all four applications.

This section provides full ESL object graphs for all four applications.

ESL Object graph for main program of POWRSWNG



The ESL object graph for subprogram MAIN_LOOP



The ESL object graph for subprogram INNER_LOOP

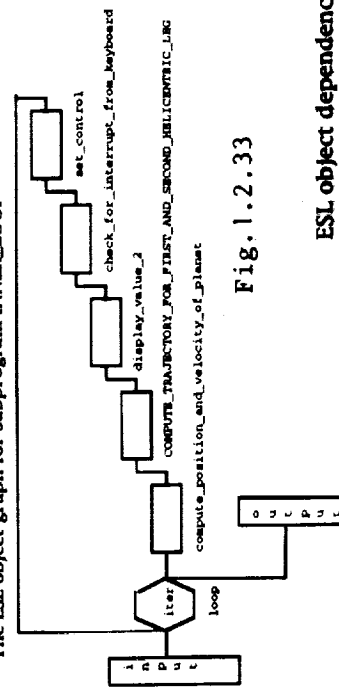
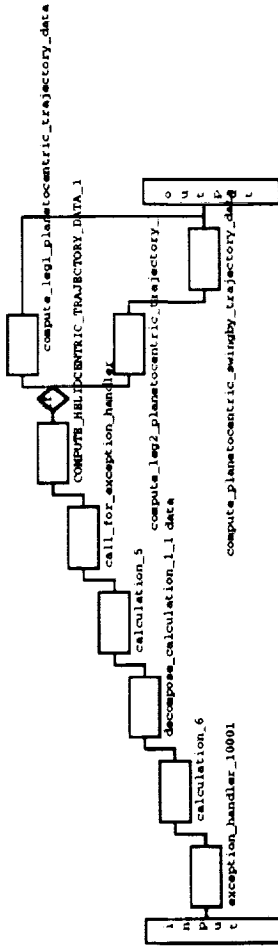
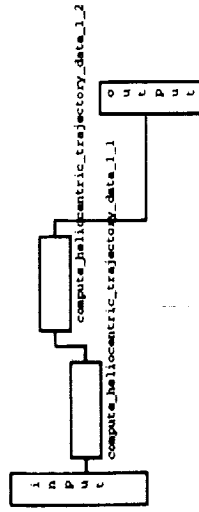


Fig. 1.2.33

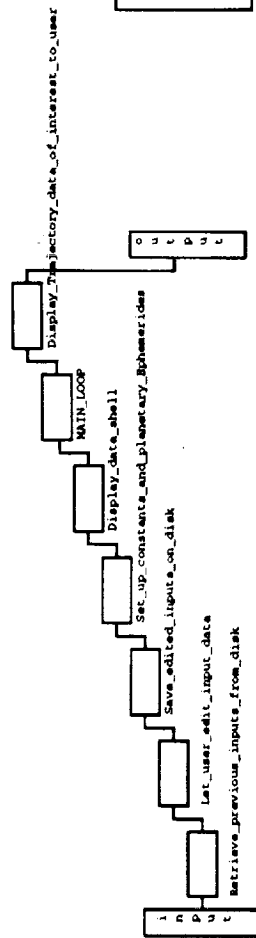
ESL object graph for COMPUTE_TRAJECTORY_FOR_FIRST_AND_SECOND_HELIOCENTRIC_LEG for POWRSWNG



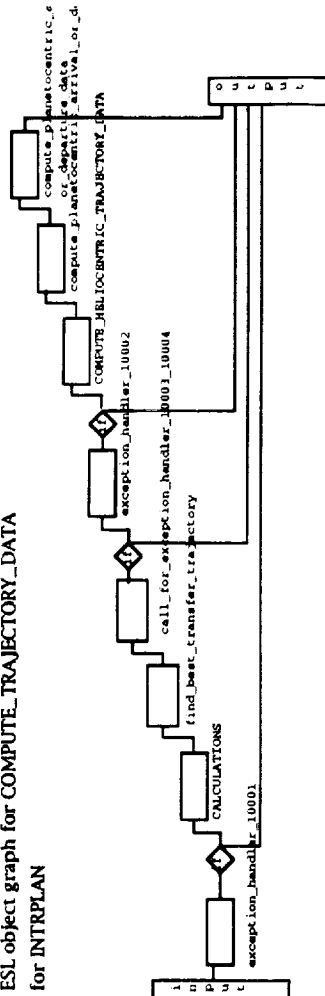
ESL object graph for COMPUTE_HELIOCENTRIC_TRAJECTORY_DATA_1



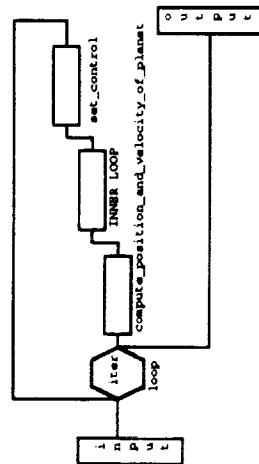
ESL Object graph for main program of INTRPLAN



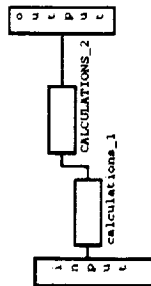
ESL object graph for COMPUTE_TRAJECTORY_DATA for INTRPLAN



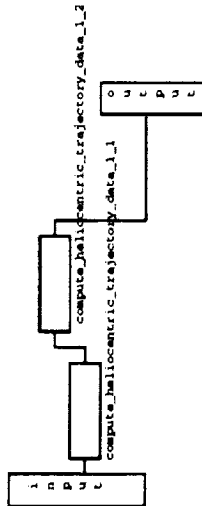
The ESL object graph for subprogram MAIN_LOOP



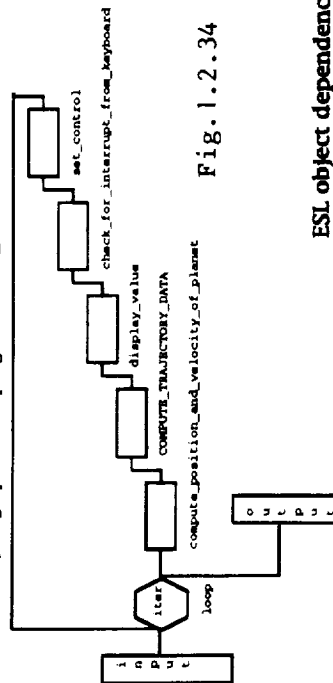
ESL object graph for subprogram CALCULATIONS



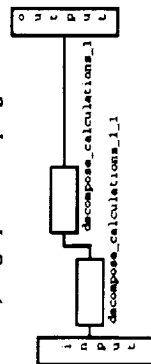
ESL object graph for COMPUTE_HELIOCENTRIC_TRAJECTORY_DATA



The ESL object graph for subprogram INNER_LOOP



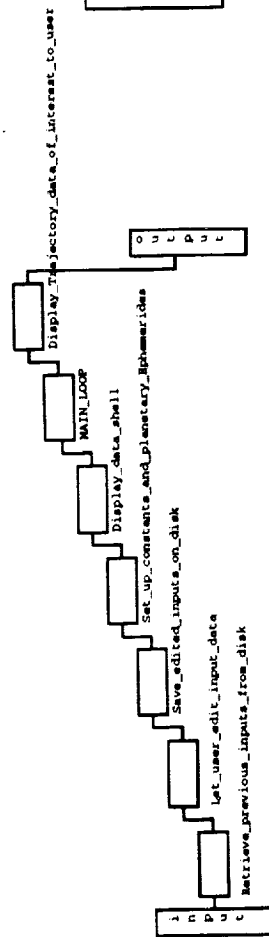
ESL object graph for subprogram CALCULATIONS_2



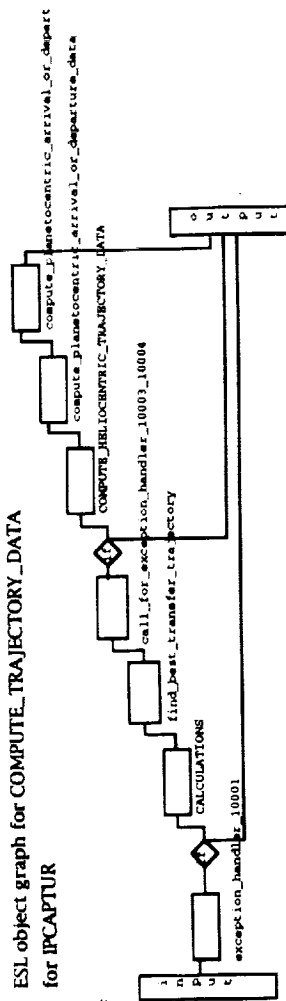
ESL object dependency graph for INTRPLAN

Fig. 1.2.34

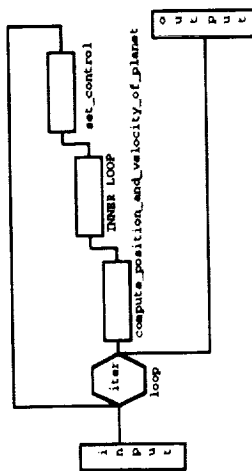
ESL Object graph for main program of IPCAPTUR



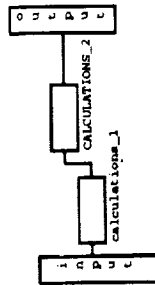
ESL object graph for COMPUTE_TRAJECTORY_DATA for IPCAPTUR



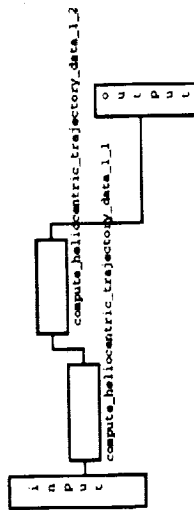
The ESL object graph for subprogram MAIN_LOOP



ESL object graph for subprogram CALCULATIONS



ESL object graph for COMPUTE_HELIOCENTRIC_TRAJECTORY_DATA



The ESL object graph for subprogram INNER_LOOP

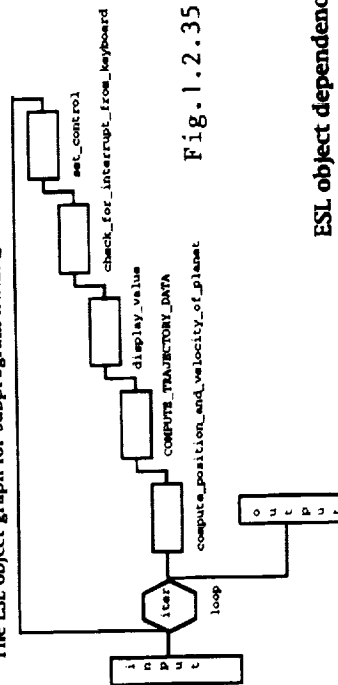
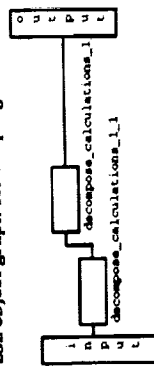


Fig.1.2.35

ESL object graph for subprogram CALCULATIONS_2



ESL object dependency graph for IPCAPTUR

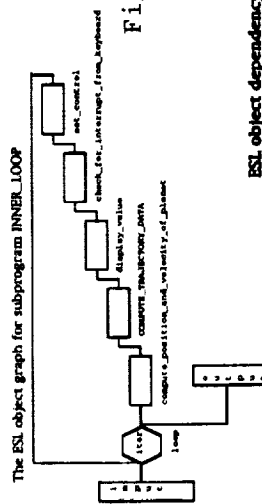
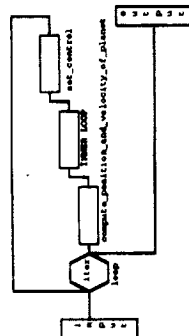
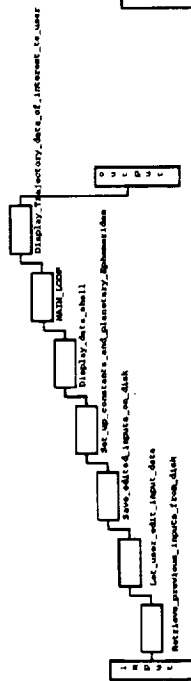
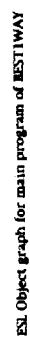


Fig. 1.2.36

ESL object dependency graph for BEST1WAY

2.1 Lessons Learned About the Current ESL Tool

- ESL graphs are similar to data flow diagrams, but also have control flows and two additional constructs: loop and if-then.
- ESL does not currently have any configuration management functions or any verification and validation functions.
- Current attributes of the components are insufficient; particularly, there are no attributes (slots) for providing the purpose of a component.
- There are several important reuse questions that must still be answered before ESL can be a truly effective reuse system. What standards should reusable components meet before they are accepted as ESL components? What makes a component reusable? As ESL is targeted toward an engineer rather than a programmer, should not the engineer see mostly domain-oriented components instead of computer-science oriented components while searching the library of components?
- FOR loops are not supported, only WHILE loops. In the ESL system, the looping structures are restricted only to WHILE loops. This is a severe draw back. It was found that, during the reengineering process, all the FOR loops in the subprograms had to be changed into WHILE loops. In order to do this, more modules had to be created (because ESL only connects modules and does not allow for the direct insertion of even small pieces of code); specifically, an unnecessary module had to be created to increment the iterated value for the new WHILE loop. If there were ESL facilities to implement FOR loops, this unnecessary creation of modules could have been avoided.
- Another shortcoming found in the ESL graphs was the lack of directional labelling. For example, there are two directions emerging from an IF node: THEN and ELSE. The ESL graphs do not label these directions, causing great confusion.
- A few serious syntax errors were found in the generated Ada code.

- When an existing graph is modified, the user is given two options when exiting the graph: save or delete. There is no option to exit the graph without making any permanent changes to it. Furthermore, if the delete option is selected, the graph is deleted from the knowledge base, but the associated drawing file, called "graphname.dwg," is not removed from the file system.
- Drawing files are not always kept consistent with the knowledge base.
- The graph editor panel cannot be resized.
- The connector drawing algorithm is too simple and needs to be improved.
- When a node is deleted on the drawing panel, that object still appears on the "Node on graph" panel.

